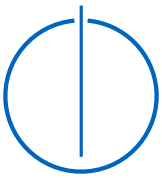
Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

An Optimized Intermediate Representation for Binary Rewriting at Runtime

David Hildenbrand



Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Eine optimierte Zwischensprache für Binärumschreibung zur Laufzeit

An Optimized Intermediate Representation for Binary Rewriting at Runtime

Author:	David Hildenbrand
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Alexis Engelke, M.Sc.
Submission Date:	15 June 2019

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

David Hildenbrand

Acknowledgments

First, I would like to thank my thesis advisor, Mr. Alexis Engelke, and my supervisor, Prof. Dr. Martin Schulz, for all the invaluable and professional discussions, ideas, and feedback. Their guidance and support contributed a lot to the result of this thesis. I owe particular thanks to Mr. Engelke for dedicating so much time.

I would also like to thank Dr. David Alan Gilbert for proofreading this thesis. His feedback greatly improved the quality of this work.

Special thanks to my employer Red Hat (most notably my managers Mr. Hai Huang and Mr. Luiz Capitulino) for enabling me to follow my dreams, giving me maximum flexibility throughout my studies, and always keeping me motivated. Working in such a wonderful environment is a privilege.

Also, my research would have been difficult without the support of my loving parents, siblings, and friends. I am extremely fortunate to have such amazing people in my life.

Most importantly, I must express my very profound gratitude to my partner Katarina for supporting me throughout these years, for always believing in me, and for being my anchor in life's ocean. You are my best friend.

Abstract

Application-guided dynamic binary optimization already turned out to be a powerful approach to increase the single-thread performance of High Performance Computing (HPC) applications. Selected binary code can be optimized and specialized at runtime to incorporate information known at runtime. Previous work performed binary code optimizations using partial evaluation as well as reusing the LLVM compiler infrastructure. While using LLVM improved the performance of the rewritten binary code, rewriting times increased, too. However, as the rewriting is performed at runtime, the rewriting time has a significant impact on the overall performance of the application. Binary code modifications at runtime require a more powerful, yet efficient, Intermediate Representations (IRs) than pure machine code.

To create an optimized machine-level IRs for efficient optimization of binary code at runtime, existing IRs in binary rewriters are analyzed. Different analyses and optimizations are implemented using this optimized IR in a prototype binary rewriting system called *Drob*. While Drob has an initial focus on x86-64, it was designed to be retargetable. Benchmark results show that Drob outperforms the simple partial evaluation approach when it comes to the runtime of rewritten binary code, however, is not able yet to optimize binary code as much as the LLVM-based approach in most cases. Although rewriting times increased compared to the simple partial evaluation approach, they are still a factor of magnitude faster compared to the LLVM-based approach.

Contents

1 Introduction

In High Performance Computing (HPC) applications, speed and scalability usually have high priority. Developers spend a significant amount of time trying to make code run faster and scale better. While handling and optimizing for high levels of parallelism is one challenge, improving the efficiency of single threads is another important challenge. It allows for better utilization of available compute resources and higher energy efficiency.

Optimizing code manually on assembly level may result in very fast code; however, this handcrafted binary code costs much development effort and is also very error-prone. While higher-level programming languages allow developing applications significantly faster with a smaller error rate, the ability to finetune code like on assembly level is lost. Hence, there is usually a tradeoff between fast code and development effort. Any mechanism to further optimize code written in high-level languages is in general of interest.

Interpreted programming languages are usually avoided in the context of HPC due to the high overhead of the interpreter. Similarly, techniques that generate code dynamically at runtime using a Just In Time (JIT) compiler are rarely used in HPC applications. [119]

Static compilers can spend much time and effort optimizing code exactly once before the application runs. JIT compilers have to perform the same tasks as fast and efficiently as possible at runtime. The resulting binary code executes in general slower, and the overhead of the JIT compiler always exists. Every node in an HPC cluster might generate the same binary code at runtime, eating precious Central Processing Unit (CPU) cycles.

Instead, programmers use statically compiled languages such as C, C++, and Fortran. As binary code generation and optimization take place at compile time, the compiled code is not able to exploit new hardware features or other additional information available at runtime. For example, information available at runtime can include computation results, data from external sources, data layout, data distribution, and available CPU features. The binary code in shared libraries ages with new CPU generations, even when used in new applications. Proprietary libraries cannot be recompiled. The source code might not be accessible.

Rewriting binary code instead of generating new binary code from higher-level languages or lower-level Intermediate Representations (IRs) has the benefit that also existing binary code with inaccessible source code can be optimized. No special preparations are needed; new languages and language extensions can be avoided. A binary rewriter can reuse the work performed by the original compiler to minimize the optimization time at runtime.

Different techniques were invented and implemented over the years to further optimize existing binary code either statically before the code is executed — Static Binary Optimization (SBO) — or dynamically at runtime – Dynamic Binary Optimization (DBO). Approaches applied at runtime have one main advantage: they can consider information available only at runtime when optimizing, similarly to dynamic code generation.

1.1 Motivating Example

Multiplying two matrices with dimensions $(n \times n)$ can be implemented trivially using the standard algorithm for matrix multiplication or the Strassen algorithm, resulting in a runtime of $\Theta(n^3)$ and $\approx \Theta(n^{2.8017})$, respectively. If it is known, that one of both matrices is the unit matrix or the zero matrix, calculating the resulting matrix is as easy as copying either the second matrix or the zero matrix. Both specialized algorithms result in a runtime of $\Theta(n^2)$. In general, when it is known that one matrix is sparse, it is usually faster to start with a zero matrix and to then calculate only the selected, relevant entries. Creating specialized algorithms is often possible if some input parameters can be assumed to be fixed, e.g., by specializing on non-zero matrix entries.

While static compilers already try to create specialized variants, this is not always possible (e.g., if input parameters are only known at runtime) or desired (e.g., code size). Therefore, it sometimes makes sense to invest some time to create a specialized algorithm at runtime. Assuming that the specialized algorithm is reused x times, the runtime of the algorithm before the optimization is t_{old}, the runtime after the optimization is t_{new}, and it takes t_{opt} to optimize, the approach can be considered beneficial if:

$$t_{opt} + x * t_{new} < x * t_{old} \tag{1.1}$$

There are examples (such as stencil computations as discussed in Chapter 8) where the optimization is already beneficial for $x := 1$. As long as $t_{new} < t_{old}$, the optimization time determines if the optimization is beneficial for $x := 1$.

While the matrix example is extreme, and such involved optimizations are already hard to realize in static compilers, this showcases the optimization potential at runtime, after compilation.

1.2 Binary Rewriting before Runtime

Compilers, especially with inbuilt link-time optimizers, can already optimize and specialize binary code statically across object files; however, they cannot optimize calls into shared libraries, and they can only use limited runtime properties when optimizing. Runtime information can only be considered to some extent by profiling the application and providing this data to the compiler. However, profiling data only represents hints, no guarantees, and is often only used to improve the code layout. SBO suffers from similar issues, but can at least optimize shared library calls and binary code with inaccessible source code, by working directly on binary code.

In general, static rewriting approaches do not know which parts of an application are worth being optimized; they have to take care of possible code size explosion, and they cannot use real runtime information when optimizing. The optimized binary code has to produce the same result as the original binary code for any inputs, resulting in little optimization opportunities.

1.3 Binary Rewriting at Runtime

Various approaches to optimize the performance of applications by rewriting binary code transparently at runtime were explored over the years; in general, there are many cases where the additional overhead harms performance instead of improving it. Two main problems are that binary code might get optimized that is not worth being optimized (e.g., little optimization potential), and that there is always an overhead involved (e.g., for profiling) when rewriting transparently.

> "When compared with traditional off-line feedback-directed optimization ... it appears that dynamic binary optimization holds few or no performance advantages." [103]

In contrast, letting an application specify which binary code to optimize, along with runtime properties, avoids having to guess in a binary optimizer about "what to rewrite" and "which properties to exploit when rewriting". An application usually knows these things better; in *application-guided dynamic binary optimization*, it specifies the binary code to optimize in the form of a binary function that has a known function signature.

The first research idea used partial evaluation to rewrite binary functions at runtime, implemented in a binary rewriting system called *DBrew*. The benchmark results showed that optimizing binary code using this approach can result in significant performance improvements; however, apart from constant propagation and full loop unrolling, DBrew missed many other optimization opportunities. [119]

Instead of using partial evaluation, the idea of Engelke et al. [35] was to lift the binary code to *LLVM IR*, a high-level architecture independent IR, and to perform optimizations on this abstracted level using the basic LLVM JIT compiler framework. This approach is referred to as *DBrew-LLVM* in this work. While the binary code generated by DBrew-LLVM is in most cases faster than the binary code produced by DBrew, the rewriting time increased massively.

1.4 An Optimized Intermediate Representation

While the LLVM JIT compiler is very flexible, it is suboptimal for rewriting existing binary code efficiently at runtime. Converting binary code to a high-level architecture independent IR results in discarding much work performed by the original compiler. Also, many architecture-specific details, such as instruction side effects or register fragments, complicate the translation to the IR; representing these details in a high-level IR requires many instructions. When generating new binary code from such a representation, new instruction selection and new register allocation are necessary; unused inserted code (e.g., to model irrelevant side effects of instruction) has to be optimized out again. In summary, the binary rewriter performs many tasks that are unnecessary.

The IR is the central piece of a binary rewriter; analyses and optimizations operate on it. When the runtime of the rewriting process is important, the binary rewriter has to use an IR that allows for efficient translation between binary code and the IR and still enables powerful analyses and optimizations.

This thesis describes an optimized IR and a binary rewriting approach for application-guided dynamic binary optimization. The approach was implemented in a prototype binary rewriting system called *Drob*, with an initial focus on x86-64[1]. Benchmark results show that Drob outperforms DBrew when it comes to the performance of the rewritten binary code; however, Drob needs more time for the rewriting process. In most cases, binary code generated by Drob cannot achieve the performance of binary code generated by DBrew-LLVM yet. However, Drob rewrites binary code an order of magnitude faster than DBrew-LLVM; more analysis and optimization passes might be able to further improve the performance of the binary code generated by Drob in the future, while still needing less time for the rewriting process than DBrew-LLVM.

1.5 Contributions

Key contributions of this thesis include:

- A survey, including an analysis, of existing binary rewriters and IRs.
- An IR designed for efficient rewriting and optimization of binary code at runtime with an initial focus on x86-64.
- A register liveness analysis, a stack analysis, and lightweight optimizations, developed in the context of the optimized IR.
- A system for rewriting, optimizing, and specializing binary functions at runtime with an initial focus on x86-64, usable to implement new analyses and optimizations.

1.6 Outline

Chapter 2 introduces some basic terms and definitions. Related work is explored in Chapter 3, giving a more detailed explanation of the general background. Existing binary rewriters and IRs are analyzed in Chapter 4. Chapter 5 discusses the basic rewriting approach, including the optimized IR. Analyses and optimizations built on top of these core components are explained in Chapter 6 and Chapter 7, respectively. Chapter 8 compares the selected approach against previous work. Chapter 9 summarizes the approach and the results, listing some possible future work.

[1] All further references to the x86-64 architecture in this thesis are based on the Intel® 64 and IA-32 Architectures Software Developer's Manual [58], unless otherwise stated.

2 Terms and Definitions

This chapter introduces some fundamental terms and definitions that are used throughout this thesis.

2.1 Binary Code

Binary code is architecture-specific machine code understood by a CPU, without any additional metadata. The semantics are described in the corresponding Instruction Set Architecture (ISA). A CPU can start executing a stream of binary code, executing the underlying program.

In contrast, bytecode, as used by the Java Virtual Machine (JVM) and other high-level language Virtual Machines (VMs), also contains low-level instructions, but usually also includes "a rich collection of metadata" [103], necessary to load and execute a program.

2.2 Intermediate Representation

The term Intermediate Representation (IR) was initially introduced in the context of compilers, whereby an IR is an internal representation of a program created during the compilation, used for analyses, transformations, optimizations, and binary code generation [111].

"*Intermediate*" highlights the fact that such a representation is only a mean to achieve a bigger goal; such a representation is different from the source (e.g., source code) format and the destination (e.g., compiled binary) format. The design of an IR depends on the intended use case.

Modern compilers use multiple levels of IRs, usually starting with a fairly high-level representation of the program and ending with a low-level representation similar to machine code. In theory, an IR can be any representation of a program reaching from syntax parse trees to graph-based IRs containing pseudo-assembly instructions, and there are various approaches to classify them. [111]

Nowadays, IRs are also used in other areas than classical static compilers. Typical use cases include dynamic compilation (e.g., LLVM IR in the JIT of LLVM [71]), binary translation (e.g., TCG in QEMU [14]), binary analysis (e.g., REIL in BinNavi [34]) and binary rewriting (e.g., VEX in Valgrind [102]).

2.3 Basic Block

A basic block is a code sequence with a single entrance into the block at the beginning of the block and a single exit at the end of the block. Other side entrances or side

exits are not allowed. Consequently, basic blocks do not allow multiple conditional and unconditional branch instructions in the same basic block. [103]

2.4 Superblock

A superblock is a variant of a basic block with less restrictions: multiple side exits are allowed. Therefore, a superblock can contain multiple conditional branch instruction along with at most one unconditional branch instruction. [56, 103]

2.5 Control Flow Graph

In a Control Flow Graph (CFG), nodes represent code sequences — e.g., basic blocks or superblocks — and the connecting edges represent possible control flow due to control flow instructions contained in the nodes [103]. CFGs are therefore not aware of the higher-level concept of functions and cannot model a call graph.

2.6 Interprocedural Control Flow Graph

In contrast to CFGs, Interprocedural Control Flow Graphs (ICFGs) can represent multiple functions (procedures). An ICFG can be considered a graph of graphs. The outermost graph describes the call graph, whereby nodes represent functions, connected by edges due to function call instructions. Each such function node represents another graph, basically a CFG. [60]

3 Background and Related Work

Application-guided dynamic binary optimization overlaps with many different research areas. This chapter gives an overview of related work, explaining the background. Less relevant work is only discussed shortly. Static code generation and general compiler optimizations are not covered.

3.1 Static Binary Analysis

Static Binary Analysis (SBA) is heavily used for security research (e.g., reverse engineering). The goal of SBA is to extract information and properties — such as higher-level semantics, information about control flow, and data dependencies — from binary code without executing it. SBA does not rewrite binary code.

Most analysis tools lift binary code to special IRs that allow for advanced analysis. Each IR is usually optimized for a specific analysis task. Examples of binary analysis framworks are BitBlaze [105] with the IR VINE, BinNavi [34] with the IR REIL, BAP [20, 62] with the IR BIR, and BARF [48]. REV.NG [32] and bin2llvm [64] use LLVM IR internally; work by Florian Märkl [84] also studied how well LLVM IR is suited for binary analysis. A paper by Kim et al. [62] gives a more detailed overview of binary analysis tools and the IRs they use.

3.2 Static Binary Rewriting

Static Binary Rewriting (SBR) modifies existing binary code before its execution. The modified binary code is saved in a new binary executable for later execution. It faces similar problems as SBA when it comes to extracting information from binary code, to perform only valid code modifications. Typical use cases include optimizing binary code, security hardening of binary code, and embedding instrumentation code into binary code.

3.2.1 Static Binary Optimization

There are many Static Binary Optimization (SBO) systems for various architectures that operate either at link-time or post-link-time. While link-time optimizers work on object files and usually have relocation information available, post-link-time optimizers work directly on binaries and might not have access to any metadata, because compiler and linker conventions might not be preserved. Optimizations might either target the runtime of binary code or the code size. Often, profiling data is used to guide optimizations.

Link-time optimizers that try to improve the performance of binary code include OM [110] for Reduced Instruction Set Computer (RISC) architectures like MIPS, ALTO [87]

for Alpha, PLTO [99, 100] for x86, ILTO [104] for IA-64, and PROPAN [59] for irregular architectures like Digital Signal Processors (DSPs). DIABLO [114] is a link-time binary rewriting framework that supports various architectures and has a focus on code size optimizations; however, it can also improve the runtime of binary code.

Post-link-time optimizers that try to improve the performance of binary code include Spike [27] for Alpha/NT, Etch [97] for x86, Ispike [79] for IA-64, and work by Henis et al. [49] for PowerPC, System 390, and x86. A more recent post-link-time optimizer for x86-64 and ARM64 is BOLT [93], achieving significant performance improvements. SecondWrite [8,9,68], a static binary rewriter based on LLVM IR, can optimize executables by automatically detecting and parallelizing loops.

Squeeze [31] and Squeeze++ [29] mainly focus on code size optimizations at post-link-time.

Vulcan [108] is an optimization framework targeting multiple architectures (e.g, x86 and IA-64), able to rewrite binaries statically and dynamically. Its focus is on optimizing and running applications in heterogeneous environments.

The IBM Automatic Binary Optimizer for z/OS [67] is a SBO system that tries to improve optimizations by recovering high-level information about the original COBOL program from binary code.

3.2.2 Static Binary Instrumentation and Security Hardening

Security hardening approaches often use Static Binary Instrumentation (SBI) to insert instrumentation code into executables or perform other modifications of executables to improve the security. Approaches that heavily rely on compiler support are not covered.

Zipr [47] can rewrite binaries to implement advanced security features such as stack randomization or dynamic canary randomization. Its successor, Zipr++ [50], focuses on stack unwinding for exception handling. Prasad et al. [95] implemented a static binary rewriter for x86 which can be used to detect stack-based buffer overflow attacks.

MULTIVERSE [13] and REINS [118] are SBR systems, developed in the context of security hardening, that have a focus on rewriting completely stripped binaries. Ramblr [115] focuses on binary code patching. STIR [117] is a SBR system that implements randomizing addresses of basic blocks at load time. SecondWrite [9, 92] can also insert instrumentation code into binaries.

Atom [109] is an old framework based on OM [110], which can insert instrumentation code into binaries at link-time. Other rewriters, such as PSI [122] and UROBOROS [116], implement similar approaches at post-link-time.

3.3 Just-In-Time Compilation

Just In Time (JIT) compilation, one form of dynamic code generation, generates new binary code at runtime, for example, by translating higher-level code to binary code. The input could be provided in the form of ordinary programming languages (e.g., Javascript) or IRs similar to the ones used in compilers (e.g., LLVM IR). JIT compilation is a huge research area. Therefore, the focus is primarily on approaches that deal with application-guided optimization and specialization.

Work by Auslander et al. [10] lets the user annotate code in C programs; for this annotated code, holes in pre-compiled machine-code templates are dynamically filled out with runtime constants. Similarly, DyC [41, 42] lets the user annotate C code that should be specialized by a JIT compiler at runtime. 'C [36] is an extension of C that allows specifying C code that is dynamically generated and can use runtime information.

deGoal [24] allows including a JIT compiler into C applications. A RISC-like language describes "compilettes" for which a JIT compiler generates binary code at runtime. deGoal statically encodes these compilettes in a custom IR, which the JIT compiler uses for dynamic compilation.

All described systems cannot deal with existing binary code, require special compilation environments or language modifications to work, and generate new binary code at runtime.

3.4 Dynamic Binary Rewriting

Dynamic Binary Rewriting (DBR) modifies existing binary code at runtime. The existing approaches can be classified into three categories by their purpose: instrumentation, translation, and optimization. Some more generic DBR systems (like DynamoRIO [19]) can be used for multiple purposes.

3.4.1 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) systems focus on inserting new code into existing binary code at runtime. In contrast to SBI systems, also dynamically generated code and self-modifying code can be instrumented. Some DBI systems try to optimize the modified binary code to reduce the runtime overhead.

Detours [54] can, for example, instrument arbitrary Win32 functions on x86. DynInst [21] provides an architecture independent Application Programming Interface (API) to insert instrumentation code at arbitrary points in a running program. Both systems use trampoline functions that save and restore all registers before the instrumentation code is executed, reusing most original binary code. METRIC [83] uses DynInst to instrument all instructions that access memory.

DynamoRIO [19], initially designed for binary optimization, evolved to a general-purpose DBR system; it can, therefore, also be used for instrumentation. HDTrans [107] is a very light-weight DBI system for x86, minimizing the runtime overhead by avoiding expensive code optimizations. BIRD [89] combines static and dynamic disassembling and instrumentation to reduce the runtime overhead.

Pin [78] focuses on efficiency by using a JIT compiler to modify and optimize the binary code. The JIT compiler uses an IR that is very similar to the underlying ISA. PinOS [22] is an extension of Pin to instrument operating system kernels. Valgrind [91] uses an architecture independent IR for its JIT compiler, on which plugins operate to build various Dynamic Binary Analysis (DBA) tools. Follow-up work by Cabecinhas et al. [23] tried to improve the performance of the binary code generated by Valgrind.

More recent research projects include DBILL [80], a cross-ISA DBI system based on QEMU [15] and LLVM, and RL-BIN [82], a DBI system similar to DynamoRIO and Pin, with a focus on very low runtime overhead when analyzing Control Flow Integrity (CFI).

3.4.2 Dynamic Binary Translation

Dynamic Binary Translation (DBT) systems translate binary code transparently at runtime from one ISA to another ISA. In contrast to interpreters, the translation of whole code blocks can be cached and reused to improve the performance and to reduce the runtime overhead. One famous historical example is MIMIC [85], a System/370 simulator for RISC. This work only considers native-to-native DBT, whereby the source and target ISA are basically the same. Other approaches are not discussed.

QEMU [15] and DisIRer [55] are DBT systems that support various frontends (source ISAs) and backends (target ISAs). They use architecture independent IRs and can be used for native-to-native DBT quite easily. However, these approaches usually result in much runtime overhead and bad runtime performance compared to native binary code.

Much research focused on improving the performance of DBT, especially related to QEMU. Examples include LnQ [52], PQEMU [33] and HQEMU [51]. Work by Chipounov et al. [26] proposed adding an LLVM backend to QEMU to generate optimized target binary code.

DynSec [94] implements another use case of native-to-native DBT; it uses an existing binary translator to dynamically apply security patches in the translation cache without modifying the original binary code, effectively rewriting the binary code.

3.4.3 Dynamic Binary Optimization

Dynamic Binary Optimization (DBO) systems optimize existing binary code at runtime. These systems either optimize the binary code transparently to the target application or the application guides the optimization process.

Transparent Dynamic Binary Optimization

Transparent dynamic binary optimizers rewrite the binary code of the target application transparently at runtime. Once activated, the full instruction stream of that application is a possible optimization target. Usually, only frequently executed instruction sequences are optimized. The remaining instruction sequences are either interpreted or executed almost unmodified. Detecting such “hot” instruction sequences and creating and managing optimized binary code usually results in quite some runtime overhead.

Dynamo [12] is a transparent dynamic binary optimizer based on a native-to-native interpreter for PA-RISC. Frequently executed instruction sequences are detected by the interpreter and optimized transparently to speed up the interpretation process. DynamoRIO [18] is the x86 variant of Dynamo; it avoids expensive interpretation by caching basic blocks and reusing them. Its API [17] for binary code modifications can, for example, be used to optimize instruction sequences.

Wiggins/Redstone [30] is a transparent dynamic binary optimizer for Alpha that automatically optimizes, reorganizes, and specializes frequently executed instruction sequences; it uses hardware-based sampling to detect hot instruction sequences. Adore [77] is a similar system for IA64. Optimizations are only applied to hot instruction sequences which are identified using performance monitoring hardware. It relies on some cooperation with the original compiler to reserve spare registers needed to perform optimizations.

Mojo [25] is a transparent dynamic binary optimizer for x86, relying on Vulcan [108], with a focus on multi-threaded applications and exception handling.

Padrone [96] is a "toolbox to perform dynamic code transformations" [96]; it attaches a separate process to the target application process, which can inject new binary code and modify existing binary code. Follow-up work by Hallou et al. [44–46] uses Padrone to vectorize and re-vectorize existing binary code.

Kistler and Franz [65, 66] describe a dynamic binary optimization approach that uses dynamic code generation: "continuous program optimization" [65]. The optimizer generates optimized binary code for application procedures in the background; the new binary code replaces the original binary code dynamically in memory. The system does not perform the optimizations on binary code but on an input format called "Slim Binary representation" [65]. According to the authors, this approach could be extended to binary code with some limitations.

BAAR [28] implements a similar approach using LLVM IR as input format; it transparently and dynamically offloads identified instruction sequences to Intel® Xeon Phi ™ processors. Theoretically, this approach could be extended to native binary code.

ExanaDBT [98] is another transparent dynamic binary optimizer for x86-64 with a focus on parallelization and loop optimizations. It automatically selects and lifts binary code to LLVM IR, where polyhedral optimizations are performed. The new binary code, generated from this representation, replaces the original binary code.

Application-guided Dynamic Binary Optimization

Yarvin and Sah proposed with QuaC [120] a C interface to a binary code optimizer for Alpha. Arbitrary binary functions can be specialized and optimized at runtime using runtime information. The rewritten binary function can be used as a replacement for the original binary function. Performance results, the prototype, and the final C interface were never published. The basic approach is very similar to the approach discussed in this work.

DBrew DBrew [35,119] — initially called Brew [119] — is a Dynamic Binary Optimization (DBO) system similar to QuaC for x86-64. Binary functions can be specialized to given runtime information using a simple C interface. A new binary function is created that can be used as a replacement for the original binary function.

DBrew uses partial evaluation to perform optimizations; it emulates the binary function instruction by instruction, starting with a given program state derived from the configured runtime information. Emulated instructions, along with information about operands, are captured. DBrew caches basic blocks that resulted from starting to emulate at a specific instruction with a spacific program state. For conditional branches, DBrew might have to follow multiple paths. DBrew creates the final rewritten binary function by properly encoding and wiring up all recorded basic blocks. [119]

DBrew implicitly propagates constants, specializes instructions to recorded operands, and removes dead code. However, DBrew cannot perform advanced optimizations because of its simple partial evaluation approach. Usually, loops are fully unrolled, eventually resulting in large binary code. [119]

DBrew-LLVM DBrew-LLVM [35] is an extension of DBrew, which uses LLVM to optimize binary code. Two optimization modes are implemented.

In the first mode, DBrew-LLVM optimizes the output of DBrew using LLVM; it lifts the basic blocks that resulted from partial evaluation to LLVM IR. The LLVM JIT is then used to generate an optimized binary function by reusing the optimization passes part of LLVM. In the second mode, DBrew-LLVM skips the partial evaluation in DBrew completely; instead, it lifts the original binary code directly to LLVM IR. Specializations and optimizations are performed via LLVM only. [35]

While DBrew-LLVM can improve the perofrmance of binary code generated by DBrew in the first mode, lifting the original binary code directly to LLVM IR results in even faster binary code in most benchmark. [35]

The downside of using LLVM is that the time needed for specializations and optimizations increased heavily. When lifting binary code to a high-level IR, side effects of the original machine instructions have to be represented in complex ways. For example, status flags, register facets, and the stack turned out to be complicated to handle. A JIT is used to generate entirely new machine code from this IR. By converting the binary code to a high-level IR and back, many optimizations performed by the original compiler are discarded, and much work is redone.

4 Survey of Intermediate Representations in Binary Rewriters

This chapter contains a collection of existing binary rewriters and IRs they use. Selected IRs are explained in more detail and the collected data is analyzed. While some flavors of binary translation (see Chapter 3) can be called binary rewriting, this analysis does not consider binary rewriters mainly used for binary translation. The data used in this survey was collected in January 2019.

4.1 Description of the Collected Data

In addition to publications and documentation, source code was also used to collect information about binary rewriters and implemented IRs. Sometimes, making a reliable statement about a certain property is impossible, because insufficient information is publicly available. In this case, either "**Unknown (?)**" is used, or an educated guess is made, annotated with "*****".

For binary rewriters, the name of the project or product is used. If a binary rewriter does not have a name, the names of the authors of the relevant publications are used. The name of an IR is assumed to match the name of the introducing binary rewriter, if no dedicated name is mentioned in the publications.

Due to limited access to details about a significant number of binary rewriters and the vast number of binary rewriters, the collected data is restricted to properties that are most interesting for creating an optimized IR for dynamic binary optimization.

4.1.1 General Information about Projects

Some information applies to IRs and binary rewriters; both are referred to as projects in this survey. Collected information about projects are the age, source code availability and if a project is still maintained. Also, used references are listed.

Year The year of the oldest identified publication related to a project is collected. If the source code indicates that a project is even older, that year is used instead. It should be treated like a rough estimate.

Source Code Availability The source code of a project is considered to be available if everybody can access it. If a publication mentions that its source code can be obtained on request, it is not freely available. The rationale is that many publications are already old, the projects are long dead and the source code is possibly lost forever. If the source

code is available, a link where the source code can be downloaded, the open-source license (if any) and the main programming language are collected. **Yes (✓)** or **No (-)**.

Maintenance A project is considered to be maintend if it has seen some activity since the beginning of the year 2017. Either a publication about the project was published, commits to an open-source project were contributed, or updates for a closed-source project were released. **Yes (✓)** or **No (-)**.

References The references, excluding source code, used to collect information about a project are listed.

4.1.2 Information about Intermediate Representations

For this survey, it was collected how an IR represents instructions and registers, and whether it allows to represent multiple functions.

Instructions The type of instruction representation used in an IR.

- **Unmodeled Machine Instructions (U)**: architecture-specific machine instructions that are not modeled in the IR (except e.g., control flow instructions); instead, they are processed in textual representation (for example by using an existing disassembler and assembler), or they are treated like binary data (e.g., merely copying them to a new location).
- **Modeled Machine Instructions (M)**: architecture-specific machine instructions modeled using some internal format, allowing them to be analyzed and modified more easily. Instructions might have been abstracted to some degree and annotated with information such as the instruction type, operands, and side effects.
- **Virtual ISA Instructions (V)**: instructions part of a virtual ISA. Usually, the instructions are architecture independent. In some cases, virtual ISAs contain also architecture-specific parts or are developed to handle a specific architecture.

Registers What type of registers an IR uses. Some publications use the terminology "variables" instead of registers. In this survey, both terms are used interchangeably.

- **Machine Registers (M)**: architecture-specific machine registers.
- **Virtual Registers (V)**: virtual registers or variables.
- **Virtual SSA Registers (S)**: virtual registers or variables that may only be assigned once — Single Static Assignment (SSA).

Functions Whether multiple functions can be modeled in an IR. **Yes (✓)** or **No (-)**.

4.1.3 Information about Binary Rewriters

Collected information that only applies to binary rewriters are the usage, the use case, and the supported architectures.

Usage How a binary rewriter and the rewritten binary code is used.

- **Static (S)**: the binary code is read from and written to an executable or libraries.
- **Dynamic (D)**: the binary code is read from and written to the address space of the current process, or of another running process.

Use Case The main use cases a binary rewriter was developed for.

- **Runtime Optimization (OPT)**: rewrite to optimize the performance of binary code.
- **Size Optimization (SIZ)**: rewrite to reduce the size of binary code.
- **Security Hardening (SEC)**: rewrite to improve or assure security.
- **Instrumentation (INS)**: rewrite to instrument binary code.
- **Patching (PAT)**: rewrite to patch binary code (e.g., to apply security patches).

Supported Architectures Architectures a binary rewriter supports or once supported, limited to two examples. Sometimes, publications are not completely clear about the actual architecture that is supported. For example, x86 might be mentioned although the rewriter only supports x86-64. Registers and instructions from examples were used to identify the supported architectures in case the source code is not available.

4.2 Collected Data

Data for this survey was collected about all identified binary rewriters. Especially hobby projects, less important scientific projects, and commercial products without scientific publications might be missing.

Table 4.1 lists all IRs in chronological order that the identified binary rewriters reuse from compilers and binary translators. Table 4.2 lists all identified binary rewriters in chronological order. If the name of an IR is given (e.g., VEX), the binary rewriter introduced this IR; information about the IR is provided. If the name of an IR is "=", the name of the binary rewriter matches the name of the IR; information about the IR is provided. If the name of an IR is "-", the binary rewriter has no dedicated, custom IR; it only reuses other IRs. Information about the reused IRs can be found in Table 4.1 or Table 4.2.

Table 4.1: *IRs reused in binary rewriters from compilers and binary translators.*

IR	Instructions	Registers	Functions	Year	Source Code	Maintained	References
Testarossa IL	V	V	✓	1998	✓[1]	✓	[81, 112]
LLVM IR	V	S	✓	2002	✓[2]	✓	[71, 72]
LLVM MIR[3]	M[4]	M, S	✓	2002*	✓[5]	✓	[2, 3]
TCG	V	V	-	2009	✓[6]	✓	[14, 74]
LLVM MC	M	M	-	2009	✓[7]	✓	[3, 70]

[1] `https://github.com/eclipse/omr/tree/master/compiler/il`, Eclipse Public License 2.0, C++ [2] `https://github.com/llvm/llvm-project.git`, University of Illinois/NCSA Open Source License, C++ [3] LLVM MIR is only indirectly used by binary rewriters using the LLVM IR [4] Machine instruction bundles for Very Long Instruction Word (VLIW) architectures can be represented, too [5] `https://github.com/llvm/llvm-project.git`, University of Illinois/NCSA Open Source License, C++ [6] `https://github.com/qemu/qemu`, GPL v2, C [7] `https://github.com/llvm/llvm-project.git`, University of Illinois/NCSA Open Source License, C++

Table 4.2: *Binary rewriters and IRs.*

Rewriter (IR)	Reused IRs	IR Instructions	IR Registers	IR Functions	RW Usage	RW Use Case	RW Architectures	Year	Source Code	Maintained	References
OM (=)		V[8]	V	✓	S	OPT	MIPS	1992	-	-	[110]
ATOM (-)	OM				S	INS	Alpha	1994	-	-	[109]
QuaC (=)		M*	M*	✓*	D	OPT	Alpha	1994	-	-	[120]
EEL (=)		V[9]	?	✓	S	?	SPARC, MIPS	1995	-	-	[69]
Spike (=)		M	M*	✓	S	OPT	Alpha	1997	-	-	[27]
Etch (=)		?	?	✓	S	OPT, INS	x86	1997	-	-	[97]
ALTO (=)		M*	M*	✓	S	OPT	Alpha	1998	✓[10]	-	[87]
Dynamo (=)		M	M, V	-	D	OPT	PA-RISC	1999	-	-	[11, 12]
Wiggins/Redstone (=)		?	?	?	D	OPT	Alpha	1999	-	-	[30]
Detours (=)		M	M	-	D	INS	x86-64, ARM64, ...	1999	✓[11]	✓	[54]
Henis et al. (=)		U*	M*	-	S	OPT	x86, PPC, s390	1999	-	-	[49]
SASI x86 (=)		U	M	-	S	SEC	x86	1999	-	-	[38]
Mojo (=)		U	M	-	D	OPT	x86	2000	-	-	[25]
DynInst (=)		M	M	✓[12]	D	INS	x86-64, ARM64, ...	2000	✓[13]	✓	[21]
Squeeze (-)	ALTO				S	SIZ	Alpha	2000	✓[14]	-	[31]
PROPAN (=)		M*	M*	?	S	OPT	DSPs[15]	2001	-	-	[59]
Vulcan (=)		M*, V	M, V	✓	S, D	OPT, INS	x86, IA-64	2001	-	-	[108]
PLTO (=)		M	M	✓	S	OPT	x86	2001	-	-	[75, 99, 100]
ILTO (-)	PLTO				S	OPT	IA-64	2002	-	-	[104]
Squeeze++ (-)	ALTO				S	SIZ	Alpha	2002	-[16]	-	[5, 29]
Valgrind (Ucode)		V[17]	M, V	-	D	INS	x86, x86-64, PPC	2002	✓[18]	-[19]	[90, 91, 101]
DynamoRIO (=)		M	M	-	D	OPT[20]	x86-64, ARM64, ...	2003	✓[21]	✓	[17, 113]
Prasad et al. (=)		U	M	?	S	SEC	x86	2003	-	-	[95]
Ispike (=)		M*	M*	✓*	S	OPT	IA-64	2004	-	-	[79]

Continued on next page.

Table 4.2 – *Continued from previous page.*

Rewriter (IR)	Reused IRs	IR Instructions	IR Registers	IR Functions	RW Usage	RW Use Case	RW Architectures	Year	Source Code	Maintained	References
Adore (=)		M	M	-*	D	OPT	IA-64	2004	-	-	[77]
Valgrind (VEX)		V	S	-	D	INS	x86-64, ARM64, ...	2005	✓[22]	✓	[4, 91]
Pin (-)	Ispike[23]				D	INS	x86-64, ARM, ... [24]	2005	-	✓[25]	[78]
HDTrans (=)		M	M	-	D	INS[26]	x86	2005	✓[27]	-	[106, 107]
DIABLO (=)		M	M	✓	S	SIZ, INS	x86-64, ARM64, ...	2005	✓[28]	✓	[114]
Sun BCO[29] (=)	-	?	?	?	S	OPT	SPARC	2005	-	-*	[76]
CFI (-)	Vulcan*				S	SEC	x86	2005	-	-	[6]
XFI (-)	Vulcan*				S	SEC	x86	2006	-	-	[37]
PittSFIeld (=)		U	M	?	S	SEC	x86	2006	-[30]	-	[86]
BIRD (=)		U*	M*	?	D	SEC, INS	x86	2006	-	-	[89]
COBRA (=)		?	?	-*	D	OPT	IA-64	2007	-	-	[61]
PEBIL (=)		U*	M*	✓	S	INS	x86, x86-64	2010	✓[31]	-	[73]
SecondWrite (-)	LLVM IR				S	SEC, OPT	x86	2010	-	✓[32]	[9, 68, 92]
Selftrans (=)		V[33]	M	-	D	OPT	x86	2011	-	-	[88]
REINS (=)		U*	M*	?	S	SEC	x86	2012	-[34]	-	[118]
STIR (=)		U	M	-*	S	SEC	x86	2012	-	-	[117]
BinCFI (=)		U	M	-	S	SEC	x86	2013	-	-	[123]
CCFIR (=)		U*	M*	-	S	SEC	x86	2013	-	-	[121]
PSI (=)		M*[35]	M	-	S	SEC, INS	x86	2014	-	-	[122]
DBILL (-)	TCG, LLVM IR				D	INS	x86-64, ARM64, ...	2014	-	-	[80]
Padrone (=)		U*	M*	-	D	OPT[36]	x86-64	2014	-	-	[96]
Hallou et al. (-)	Padrone, LLVM IR				D	OPT	x86-64	2015	-	✓	[44–46]
IBM ABO[37] (-)	Testarossa IL				S	OPT	s390x	2016	-	✓[38]	[67]
DBrew (=)		M	M	-	D	OPT	x86-64	2016	✓[39]	✓	[119]
UROBOROS (=)		U	M	✓	S	INS	x86, x86-64	2016	✓[40]	-	[116]

Continued on next page.

Table 4.2 – *Continued from previous page.*

Rewriter (IR)	Reused IRs	IR Instructions	IR Registers	IR Functions	RW Usage	RW Use Case	RW Architectures	Year	Source Code	Maintained	References
ExanaDBT (-)	LLVM IR				D	OPT	x86-64	2017	-	✓	[98]
DBrew-LLVM (-)	LLVM IR				D	OPT	x86-64	2017	✓[41]	✓	[35]
RevARM (=)		M*	M*	?	S	SEC	ARM	2017	-	✓	[63]
Zipr (=)		U*	M	-*	S	SEC	x86, x86-64	2017	-	✓	[47]
Zipr++ (-)	Zipr				S	SEC	x86, x86-64	2017	-	✓	[50]
RL-BIN (=)		?	?	?	D	SEC	x86	2017	-	✓	[82]
Ramblr (=)		U	M	-*	S	PAT	x86, x86-64	2017	-	✓	[115]
MULTIVERSE (=)		U	M	-*	S	SEC, INS	x86, x86-64	2018	✓[42]	✓	[13]
BOLT (=)	LLVM MC	M	M	✓[43]	S	OPT	x86-64, ARM64	2018	✓[44]	✓	[93]

[8] Designed for RISC architectures [9] "EEL instructions are abstractions of RISC-like machine instructions" [69] [10] `https://www2.cs.arizona.edu/projects/alto/Download/`, Custom License, C [11] `https://github.com/Microsoft/Detours`, MIT License, C++ [12] For snippets to be inserted and to find instrumentation points [13] `https://github.com/dyninst/dyninst`, LGPL 2.1, C [14] `http://www2.cs.arizona.edu/projects/squeeze/`, No License, C [15] Developed for irregular architectures like DSPs [16] Was located at `http://www.elis.ugent.be:80/~brdsutte/squeeze++/` [17] Contains virtual instructions tailored for specific architectures (e.g., `LEA1` for x86) [18] `http://valgrind.org/downloads/old.html`, prior to 3.0.0, GPL v2, C [19] Ucode has been replaced by VEX [20] Evolved to a more general runtime code modification system [21] `https://github.com/DynamoRIO/dynamorio`, BSD License, C [22] `https://sourceware.org/git/?p=valgrind.git`, GPL v2, C [23] Pin is based on Ispike. The amount of IR modifications is unknown [24] Support for some architectures was dropped since the publication was released [25] Commercial version available at `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool` [26] Was supposed to be a DBT system first [27] `http://srl.cs.jhu.edu/projects/index.html`, Custom License, C [28] `https://github.com/csl-ugent/diablo`, GPL v2, C [29] Sun Studio Binary Code Optimizer [30] Was located at `http://pag.csail.mit.edu/~smcc/projects/pittsfield/` [31] `https://github.com/mlaurenzano/PEBIL`, GPL v3, C [32] Commercial version available at `https://www.secondwrite.com/` [33] Designed for x86 [34] Never uploaded to `https://sourceforge.net/projects/x86reins/` [35] Based on the Intel® XED decoder. [36] Padrone is rather a platform that allows other tools to implement optimizations [37] IBM Automatic Binary Optimizer for z/OS [38] Commercial version available at `https://www.ibm.com/de-en/marketplace/improved-cobol-performance` [39] `https://github.com/caps-tum/dbrew`, LGPL v2.1, C [40] `https://github.com/s3team/uroboros`, No License, Python [41] `https://github.com/caps-tum/dbrew`, LGPL v2.1, C [42] `https://github.com/utds3lab/multiverse`, GPL v3, Python [43] The concept of functions is implemented on top of LLVM MC [44] `https://github.com/facebookincubator/BOLT`, University of Illinois/NCSA Open Source License, C++

4.3 Details about Selected Intermediate Representations

Some IRs, especially the ones reused from compilers and binary translators, are worth mentioning some further details, to clarify some of the core concepts.

4.3.1 Testarossa IL

Testarossa IL was invented as part of the Testarossa compiler by IBM when working on a commercial Java JIT implementation [81].

Nowadays, the IR is reused in other compilers and binary translators developed by IBM. IBM contributed some parts of Testarossa in 2016 to the Eclipse OMR project [112].

The IR is based on trees, whereby nodes have opcodes and produce at most one value. The children of a node correspond to the operands. Nodes (instructions) can still have side effects in the form of evaluation orders that have to be respected. As the IR originates from a compiler, it uses strongly typed variables and is able to represent multiple functions. [81]

4.3.2 TCG

TCG is the IR used in QEMU; it defines a virtual ISA that stores instructions in basic blocks and uses strongly typed (e.g., 32-bit and 64-bit integers) variables. Although the concept of functions exists, it does not correspond to functions in the original binary code. A function in TCG is a translated code block. Minor optimizations, including a simple liveness analysis, are carried out on basic blocks in QEMU. [14]

While the frontend in QEMU is responsible for converting binary code to TCG, the backend is responsible for generating new binary code from this representation. Helper functions can be called from TCG-generated code in case a source machine instruction is too complicated to be represented in TCG natively, but also if an target architecture does not implement code generation for a TCG instruction. [14]

In general, TCG was designed for correct binary translation between different ISAs, not for maximum efficiency and performance.

4.3.3 VEX

VEX is the IR introduced by newer versions of Valgrind, a framework to build dynamic analysis tools. Other projects reuse VEX; for example, *angr*, a Static Binary Analysis (SBA) framework, uses VEX to carry out its analysis. [102]

VEX defines an architecture independent virtual ISA, stores a sequence of instructions in superblocks, and uses strongly types SSA variables. It is much more similar to IRs used in compilers than to assembly-like IRs used in comparable frameworks. The “guest state”, such as CPU registers, is stored in memory and is accessed from the IR using special “get” and “put” instructions. Just like in TCG, helper functions can be called from the generated binary code to handle complicated guest instructions, but also for instrumentation purposes. [4]

4.3.4 LLVM IR

LLVM IR is the most important IR used in the LLVM compiler framework. The compiler frontend translates source code into LLVM IR. On this level, different architecture independent optimizations can be performed. The backend is responsible for generating binary code from this representation. The LLVM JIT compiler also uses LLVM IR as input for code generation at runtime. [71]

LLVM IR uses strongly typed SSA variables. While LLVM IR is light-weight and provides low-level operations, it can represent high-level features cleanly. To support high-level languages, it provides plenty of features such as a complex typing system and global variables. Also, it can represent multiple functions. [1]

4.3.5 LLVM MIR

LLVM MIR (Machine IR) is the architecture-specific representation used in the LLVM compiler framework. The LLVM compiler translates code from LLVM IR to LLVM MIR. For example, architecture-specific optimizations and register allocation are performed on LLVM MIR by LLVM. [2]

Register allocation maps SSA variables to machine registers; the IR can represent both types. Instructions are represented using an architecture-specific opcode and a list of operands. The IR stores sequences of instructions in basic blocks. Functions group basic blocks, along with other information like constant pools. [1]

4.3.6 LLVM MC

LLVM MC is another IR used in the LLVM compiler framework, introduced when including an assembler into the LLVM compiler [70]. Later, it was also used for the disassembler, the debugger and the JIT compiler in the LLVM project [43]. The LLVM code generator emits binary code from LLVM MC [70].

LLVM MC represents binary code as found in an object file, without any high-level information like global variables, constant pools or data types. Instead, the modeled entities are object-file sections, symbols (labels) and instructions. A machine instruction is represented using an architecture-specific opcode and a list of operands. For example, operands can be immediates or register identifiers. [3]

Information about functions might only be available via debug information (e.g., DWARF) like it would be stored in an object file. However, in contrast to LLVM MIR, the concept of functions does not exist in LLVM MC.

4.4 Analysis of the Collected Data

This section analyzes the collected data, considering the different collected properties of binary rewriters and IRs.

4.4.1 Usage and Use Cases of Binary Rewriters

36 out of the 58 identified binary rewriters can rewrite binary code statically and 23 binary rewriters can rewrite binary code dynamically. Vulcan [108] is the only identified binary rewriter that can be used statically and dynamically.

Most binary rewriters (27) — especially many early binary rewriters — were developed to optimize the performance of binary code. Security hardening (17) and binary instrumentation (16) are the other two important use cases of binary rewriters. Binary code size optimization (3) and binary code patching (1) is uncommon. The intended use case of EEL could not be identified.

4.4.2 Reuse of Intermediate Representations

Looking at the collected data, it is not that common to reuse IRs in binary rewriters. However, during the last couple of years more and more binary rewriters started to reuse IRs from other projects. Usually, only follow-up research projects reuse IRs from previous research projects developed at the same University or company. One exception is LLVM IR, which was reused quite often along with the LLVM JIT compiler in dynamic binary rewriters, especially with a focus on performance optimization of binary code.

Reusing IRs from DBT systems like TCG is a rare case. In general, most projects define custom IRs. Possible reasons could be that IRs are often strongly intertwined with the introducing binary rewriter and that specific architectures or analyses have particular requirements for an IR. Also, defining a new IR requires usually less work than having to work around oddities in an existing IR, especially considering x86 and x86-64.

4.4.3 Source Code Availability, Maintanance and Programming Languages

Most publications about binary rewriters describe the IR they use only with a few sentences, which is not enough to explain relevant parts in detail. Having access to source code allows researchers to get a deeper insight and to reuse ideas or even code. For binary rewriters and IRs for which no source code is available, the quality of the collected data is in most cases worse. Details are lost, and the publications are only of limited use for other researches working in the same area, trying to solve similar problems.

Source code is only publicly available for three out of nine binary rewriters that were published since the beginning of 2017. However, in contrast to early binary rewriters, there seems to be a tendency to release more source code.

The source code of two binary rewriters was once publicly available via private web sites. As the web sites are offline, the source code is also gone. Looking at the project maintenance, most projects that are hosted on external services are still maintained. Of course, there are also some projects that were abandoned over the years, but compared to projects that share source code via private web sites, it is evident that external hosting services make it much easier for other developers and researchers to collaborate. It is unknown why source code was released — maybe because the authors want other people to collaborate, or maybe because other researches requested access to the source code.

The most frequently used programming languages in binary rewriters for which source code is available are C and C++. In addition to some projects written in Python, no other

programming languages are used for the core components. No dynamic binary rewriter was written in Python.

4.4.4 Supported Architectures of Binary Rewriters

While early binary rewriters focused on RISC architectures such as MIPS and Alpha, rewriters published during the last years primarily target x86-64. Also, supporting ARM64 gets more popular. x86 is still a frequent rewriting target, although some rewriters already only support x86-64, skipping its 32-bit predecessor. Work on new binary rewriters for VLIW architectures and legacy architectures (e.g., SPARC, MIPS, Alpha, and PA-RISC) stopped; only selected rewriting systems still support some of these architectures.

Some binary rewriters such as Detours, DynInst, Valgrind, and DynamoRIO, implemented support for many architectures over the years; however, the majority only supports selected architectures. DBILL was able to support many architectures natively by reusing QEMU and LLVM.

For security hardening, binary rewriters almost exclusively target x86 and x86-64. Code patching is only implemented for x86 and x86-64. Binary code size optimizations are primarily performed on Alpha, except for DIABLO that supports multiple architectures. Most probably the size of the binary code produced by compilers is not an issue on modern architectures, and this use case is not relevant anymore. Systems for performance optimization of binary code and binary instrumentation were implemented for various architectures.

Binary rewriters that support multiple architectures use almost exclusively either virtual ISA instructions or modeled machine instructions to represent instructions in their IR. For example, BOLT, DynamoRIO, and DIABLO use modeled machine instructions and support multiple architectures. All binary rewriters that use only LLVM IR (and not TCG additonally) support only x86 and x86-64. Maybe, lifting binary code to LLVM IR is the real challenge, and this topic was not explored by researchers for other architectures yet. Another explanation could be that other architectures are just too slow to use LLVM efficiently in a dynamic binary rewriter.

4.4.5 Instructions in Intermediate Representations

Several static binary rewriters use unmodeled machine instructions in their IRs. Mostly control flow instructions are identified and processed, and other instructions are copied or handled transparently using existing disassemblers and assemblers. While this is sufficient in many cases of security hardening and instrumentation, code modifications that can be performed by binary optimizers are limited. For example, Mojo [25] only cares about control flow instructions to build and link paths of basic blocks. Similarly, the binary rewriter from Henis et al. [49] is only able to perform block layout optimizations. Optimizations on instruction level inside basic blocks require information about the semantics of instructions.

The majority of dynamic binary rewriters use modeled machine instructions in their IR. In contrast to unmodeled machine instructions, more involved analyses and code modifications are possible. The overhead of lifting binary code to a virtual ISA and to

regenerate binary code from this representation is avoided. Using a virtual ISA can make specific tasks (such as analyses, modifications, and optimizations) more comfortable to implement. For example, Ucode defines a virtual ISA and was used in Valgrind to better deal with the complexity of the x86 instruction set [101].

In the IR used by Vulcan [108], virtual ISA instructions and unprocessed machine instructions can coexist. The rewriter lazily converts the latter to the virtual ISA. As source code was never released, details about the representation of instructions in the IR are unknown.

Custom virtual ISAs are barely used in binary rewriters. Only OM, EEL, Valgrind, Vulcan, and Selftrans designed such IRs. The virtual ISA used in the IR of Selftrans [88] was designed to assist the vectorization of loops on x86 and for other architecture-specific optimizations. The publication [88] of Selftrans is not very specific about how the IR models instructions.

4.4.6 Registers in Intermediate Representations

Most IRs model only architecture-specific machine registers. When working with unmodeled machine instructions, machine registers are implied.

Virtual registers are, in general, used rarely. In case they are used, they are often combined with a virtual ISAs, especially when the IR is reused from a compiler or a binary translator. A binary rewriter can usually not insert arbitrary register spills into the rewritten binary code, assuring that the functionality of the original binary code is not changed [29]. For this reason, using virtual registers (including SSA variables) is often avoided. However, register spills are not an issue if the whole instruction stream of a target application is rewritten transparently, like done in Valgrind. The registers of the target application can be saved to different locations, avoding to modify the stack of the target application. Only three discussed IRs support SSA variables — LLVM IR, LLVM MIR, and VEX.

Dynamo and Ucode combine virtual and machine registers in the same representation; the machine registers represent a hint for register reallocation. The LLVM compiler performs register allocation in LLVM MIR, allowing machine instructions to use SSA variables before it replaces the variables by machine registers. As Vulcan lazily converts the binary code to the virtual ISA, also machine registers are effectively part of the IR until the machine instructions are converted.

4.4.7 Functions in Intermediate Representations

Many IRs used in static binary rewriters are able to represent multiple functions explicitly. Especially if binaries or object files still include metadata stored by the original compiler (e.g., debug information), information about functions is usually included and can be extracted. Often, this metadata is not available, because it is not required to execute binary code. Then, this high-level concept has to be reconstructed from binary code.

Many binary rewriters model functions in their IR only if machine instructions are modeled as well. Most binary rewriters developed for the use case of security hardening do

not care about functions. For example, all instructions of a specific type can be modified without information about functions.

IRs used in static binary optimizers — such as PLTO, ILTO, and BOLT — allow to represent multiple functions. This information can be used to perform better optimizations, e.g., to selectively inline functions.

In dynamic binary optimizers, the concept of functions is irrelevant as long as only the current stream of instructions ("traces") executed by the target application is optimized transparently. For this reason, IRs used in Dynamo, DynamoRIO, and Mojo do not represent functions explicitly.

IRs used in application-guided dynamic binary optimizers sometimes model functions explicitly (e.g., QuaC and LLVM IR) and sometimes not (e.g., DBrew). For example, DBrew effectively inlines all functions and works only on basic blocks throughout all stages of the rewriting process.

Only selected DBI systems need information about functions. If DBI systems rewrite the complete instruction stream of an application transparently, this information is not helpful. However, if selected functions are modified in-place, like in DynInst, information about instructions is necessary to correctly identify and modify all instructions.

Higher-level IRs used in compilers, such as LLVM IR, naturally model the concept of functions, so higher-level languages can be cleanly mapped to them.

Lower-level IRs used in compilers, such as LLVM MIR, still care about functions, for example to generate function prologs and epilogs, for basic block placement, and to generate debug information [16].

IRs used in assemblers and disassemblers, such as LLVM MC, do not represent functions explicitly. In object files, information about functions might only be contained in the form of debug information. This information has to be generated from higher-level IRs. LLVM uses LLVM MC to emit code instead of using a classical assembler [3]. Information about functions is discarded when lowering from LLVM MIR to LLVM MC. While debug information might still be available, functions are no longer modeled as separate entities. BOLT uses LLVM MC and has to model functions manually on top of that IR.

4.4.8 Conclusions

Binary rewriters barely reuse IRs, and if so, the reused IRs often originate from compilers (i.e., LLVM). Source code of binary rewriters, and, therefore, of the IRs they use, is often not available. Also, publications are not very specific about the details of the IRs. Reusing generic IRs is in inefficient, and requires more effort than merely introducing a new IR.

Binary optimizers mostly use architecture-specific machine instructions and machine registers in their IR. Efficiency is especially relevant for dynamic binary rewriting; in this context, virtual ISAs are usually avoided. Virtual registers are, in general, barely used.

IRs are tightly coupled to the intended use case of the introducing binary rewriter. A binary rewriter that mostly focuses on the injection of new binary code (e.g., instrumentation) has different requirements than a binary rewriter that modifies or partly removes existing binary code (e.g., optimization). For example, while static binary rewriters often allow representing multiple functions in their IR, especially dynamic rewriters that

transparently rewrite the whole instruction stream of a target application don't need this information. The majority of dynamic binary rewriters are of this kind.

For application-guided binary optimization, only a very basic machine-level IR (DBrew) and LLVM IR (DBrew-LLVM) were used so far. In contrast to transparent dynamic binary optimization approaches, application-guided binary optimization is much more similar to static binary optimization (e.g., ICFG reconstruction), however, with a focus on rewriting efficiency.

5 Approach

This chapter introduces the general rewriting approach. The basic components of the implemented prototype rewriting system are explained, along with the heart of the binary rewriter, the IR. Not covered in this chapter are any kinds of analyses and optimizations performed on the IR. These topics are discussed in Chapter 6 and Chapter 7, respectively.

5.1 Rewriting Process

Figure 5.1 shows a high-level overview of the rewriting process.

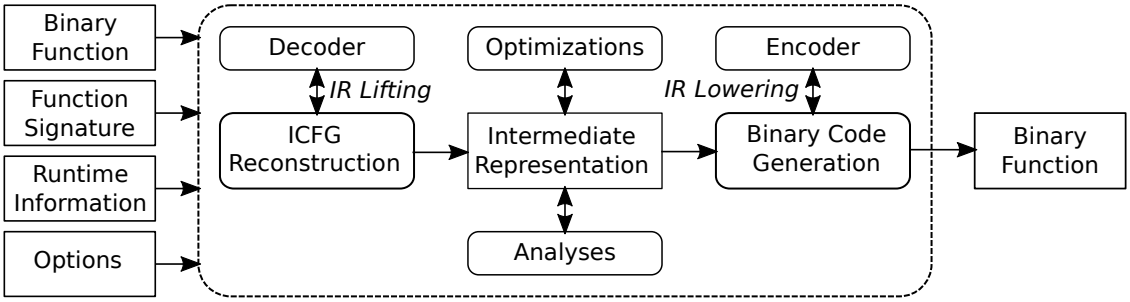


Figure 5.1: *Overview of the rewriting process.*

5.1.1 Considerations

As discussed in Chapter 4, binary rewriters are designed for particular use cases, which is why most binary rewriters use distinct IRs. The target use case and other requirements can profoundly affect the design of a binary rewriter and its IR.

Rewriting any binary code valid for a specific architecture is an ambitious goal. Especially when rewriting binary code transparently for the target application, it is usually a strict requirement that any binary code is rewritten reliably, and that the result of the binary code is not modified. For example, it is impossible for security mechanisms that use DBI to supervise applications transparently to execute an unknown instruction stream because they cannot tolerate losing control of the target binary code. However, when letting an application guide the rewriting process, this requirement can be relaxed — in particular when optimizing selected binary code. If rewriting fails, the binary rewriter can inform the application about the failure, and let it decide how to proceed. For example, the application can execute the original binary code.

In the context of this work, the focus is on dynamic optimization of binary functions produced by C compilers. Handcrafted assembly code is not of interest. The binary

rewriter optimizes selected binary functions at runtime, exploiting additional runtime information (e.g., provided by the application). The rewriting system executes in the same address space as the original and the rewritten binary code.

5.1.2 Inputs

The rewriting process has a set of required and optional inputs. The required inputs are:

1. The binary function to rewrite, which is provided via the address of the first instruction of that binary function in memory. All binary code is assumed to already reside entirely in memory in an executable-ready form, including all implicitly called functions.

2. The function signature of the binary function. Using the signature and a given calling convention, the locations (in registers and in memory) of input parameters and return values can be determined. No reliable way was identified to encode the function signature automatically in a C program at compile time, assuming debug information is not available and the symbol names of functions (e.g., created by C++ compilers for name mangeling) don't contain information about the function signature. Therefore, it has to be specified explicitly.

The function signature can be considered optional when rewriting without performing invasive code modifications. However, as the focus of this work is binary code optimization, the function signature is strictly required to identify valid code modifications. The optional inputs are:

1. Runtime information about function parameters. Examples include parameter values known at runtime, guaranteed minimum alignment of a pointer, and if values read via a pointer can be assumed to never change.

2. Runtime information about the runtime environment in general. For example, a memory range in the address space of the process that can be assumed to never change.

3. General options to configure the rewriting process. For example, how errors during the rewriting process should be handled.

5.1.3 Steps

In the first step of the rewriting process, the binary rewriter reconstructs the ICFG of the given binary function and translates it to the IR (IR lifting). The instruction decoder processes one instruction at a time, performing the translation and detecting control flow instructions. The ICFG reconstruction uses information about the control flow instructions to reconstruct the ICFG.

In the next step, various optimizations are performed on the IR. Optimizations often require analysis data. The analyses also operate on the IR directly and attach analysis data to different parts of the IR, where optimizations can access it.

The last step consists of creating the rewritten binary function, translating the IR back to binary code (IR lowering). For some instructions, the original binary code can be reused, merely copying the original binary instruction. For other instructions, the instruction encoder has to emit new binary code.

Multiple rewriting steps might need access to input data, in addition to working on the reconstructed ICFG. The function signature and runtime information are especially relevant for analyses and optimizations. All steps have access to general options.

5.1.4 Output

The output of the rewriting process is a rewritten binary function provided via the address of the first instruction in memory. If rewriting failed, depending on the configuration of the rewriting process, different actions are possible. One option is to return the original binary function, allowing the caller to continue in any case, hiding errors. Another option is to return an error (e.g., a `NULL` pointer), so the caller explicitly has to handle a failed rewriting attempt. Also, aborting the execution of the application is possible, treating the failed rewriting process as a critical error.

5.2 Restrictions

Many issues already have to be sorted out to rewrite binary code without performing invasive modifications. Therfore, taking care of many special cases is impossible and is left for future work (see Section 9.1). This section lists restrictions of the protoype rewriting system and the IR it uses.

5.2.1 General Restrictions

The instructions modeled in the IR and the specializations implemented in the prototype rewriting system were selected to successfully rewrite binary code from a benchmark that was already used to evaluate previous work (see Chapter 8). General restrictions of the binary rewriter are:

- Only x86-64 is supported. While the IR and most parts of the rewriter were designed to be retargetable, it is expected that adding support for other architectures might require extensions and modifications of the IR and core parts of the rewriting system. Especially, supporting concepts like instruction bundles (used in VLIW ISAs) and delay slots might be problematic.

- The only supported calling convention is the *System V AMD64 Application Binary Interface (ABI)* [53], because the main focus is x86-64 on Linux.

- The IR does not model privileged instructions and registers that are only accessible via privileged instructions. The focus is on rewriting userspace applications.

- Self-modifying code is not supported.

- Dynamic library loading on first function invocation prohibits a full ICFG reconstruction.
- Signals are not supported.
- For simplicity, analyses and optimizations are only performed on the entry function in the reconstructed ICFG, not on functions called by the entry function. However, called functions can be rewritten. Cross-function analyses and optimizations are left for future work. For example, conditional inlining of functions can allow analyzing and optimizing some called functions easily.
- The ICFG reconstruction is not able to handle most indirect branches and indirect function calls. Therefore, jump-tables cannot be reconstructed, however, redirecting control flow to unknown binary code is supported.
- Return Oriented Programming (ROP) and non-local jumps (e.g., jumping from a function to a location in a different function that is not the function entry point) are not supported. Binary code emitted by compilers for ordinary programs does usually not make use of these techniques.

5.2.2 x86-64 Specific Restrictions

Only a small subset of all x86-64 instructions is modeled in the IR, consisting of most control flow instructions and a selection of basic data handling instructions (e.g., `MOV`), basic arithmetical instructions (e.g., `ADD`), basic logical instructions (e.g., `XOR`), and Intel® SSE instructions. Examples of unsupported x86-64 instructions and features include:

- All instructions using `VEX` prefixes, including Intel® AVX instructions.
- Intel® MMX instructions.
- Legacy x87 Floating Point Unit (FPU) instructions, including registers.
- `LOCK`, `REPNE`, `REP` and segment override prefixes.
- Segmentation. This implies that thread-local variables are not supported for the System V AMD64 ABI [53].
- Instructions with complex side effects, such as the `INT`, `SYSCALL`, and `SYSENTER` instructions. For example, Linux uses the `SYSCALL` instructions for system calls.
- Advanced features like Intel® `MPX`.

Instructions accessing global variables via RIP-relative addressing cannot be rewritten under some conditions (see Section 5.4.8).

The IR was designed to allow modeling of most x86-64 instructions and registers in the future. Ordinary data handling, arithmetical, logical, floating-point, and vector instructions can be supported with minor IR extensions. Instructions with complex side effects and complex predicates cannot be modeled without significant modifications of

the IR. String instructions that have an unknown memory access size are problematic to represent in the IR. For most instructions, implementation and modeling effort are the limiting factors.

5.3 Intermediate Representation

This section discusses requirements and considerations for designing an optimized IR for application-guided dynamic binary optimization. An overview of the IR is given and the essential concepts are explained. The separate components of the IR are described in more detail.

5.3.1 Requirements

The following requirements were identified to be especially important:

1. **Efficient construction from binary code**: the IR should be constructible from binary code efficiently.
2. **Efficient generation of binary code**: the IR should be translatable to binary code efficiently. If possible, original binary code of selected instructions should be reused, avoiding new code generation completely, resulting in faster binary code generation.
3. **Generation of efficient binary code**: especially if there are no or little optimization opportunities, the binary code generated from the IR should not be slower than the original binary code. Minor exceptions are acceptable.
4. **Efficient emulation and analysis**: it should be possible to express all effects of an instruction, even if they have to be expressed conservatively. Emulation and analysis of single instructions should be efficient.
5. **Efficient modifications**: it should be possible to modify binary code represented in the IR efficiently.

5.3.2 Considerations

Trying to support all possible instructions is desirable. However, there are usually some instructions that are highly irregular, and many instructions are unused in practice. Excluding such instructions and focusing only on the essential instructions allows keeping the IR simpler and rewriting, in general, more efficient. One of the main goals of a DBO system should be efficiency.

If a simple IR does not allow modeling specific complex instructions, it is always an option to break these instructions up into simpler instructions, which can be represented with less effort. One example of a complicated control flow instruction on x86-64 is the `LOOP` instruction, explained in more detail in Section 5.4.5. Of course, it is not always possible to express any complex instruction using simpler instructions that are part of the same ISA, guaranteeing the same functional behavior and no undesired side effects.

One alternative is to allow unmodeled instructions in the IR; this can be done as long as a the functional behavior of these instructions does not change when relocated in the address space. On x86-64, instructions that use RIP-relative addressing cannot simply be copied to different locations. As discussed in Chapter 4, having unmodeled instructions in an IR limits analyses and optimizations.

Another alternative is to branch to an unknown instruction, essentially continuing to execute an unknown instruction stream; this also prohibits many optimizations. The binary function is only partially rewritten and still executes parts of the original code.

Last but not least, if an IR cannot support specific instructions, the rewriting process may be aborted at any time, signaling failure. Aborting may be preferred if the rewriting system is unable to perform any optimizations and eventually even generates slower binary code than the original binary code.

5.3.3 Overview

The main components of the IR are the ICFG, functions, superblocks, registers and instructions. The following sections explain these components in more detail.

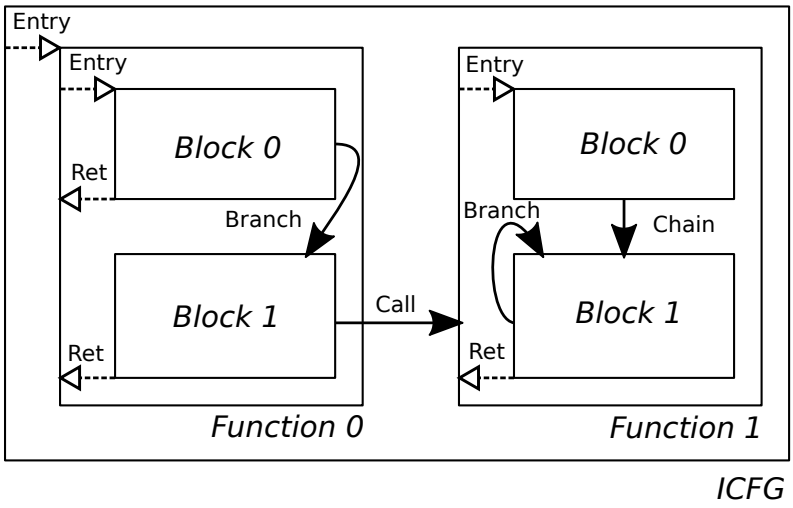


Figure 5.2: *Simplified example of a reconstructed ICFG in the IR. This example does not include instructions and registers, and only shows simplified edges.*

Figure 5.2 shows a simplified example of a reconstructed ICFG represented in the IR. The entry function (e.g., specified to be rewritten) calls a second function. Both functions contain two superblocks. Control flow instructions contained in superblocks can branch to superblocks, can call functions, and can return from the containing function. Each function has exactly one entry superblock and can contain multiple return instructions.

The IR allows representing multiple functions explicitly, to selectively inline functions and to properly detect and handle complicated rewriting situations (e.g., recursive functions and Return Oriented Programming (ROP)).

One key concept of the IR is that architecture-specific instructions and registers are used, not a virtual ISA. Instructions can either be modeled or unmodeled. Of course,

unmodeled instructions prohibit analyses and optimizations, as the semantics are unknown; however, more binary code can be rewritten natively. New instructions can be modeled in the IR incrementally, to better optimize selected binary code.

The IR provides a framework to describe architecture-specific instructions and registers, allowing the rewriter to handle them in an architecture independent way in many situations. The idea of optimizing on modeled machine instructions and registers is inspired by the dynamic binary optimizer QuaC [120], and static binary optimizers such as PLTO [99] and BOLT [93]. More details about these rewriting systems and IRs they use are discussed in Chapter 4.

Keeping instructions in the IR very close to instructions defined by the underlying architecture allows translating from binary code to the IR and back efficiently. Binary code generated from the IR without any optimizations will, in most cases, resemble the original binary code, implying similar performance.

Allowing to accurately specify information about architecture-specific registers and instructions in a simple, generic way, enables to analyze instructions efficiently. The emulator can often reuse native machine instructions to compute the effects of IR instructions. In addition, the binary rewriter can sometimes reuse the original binary code of instructions, avoiding new code generation.

5.3.4 Interprocedural Control Flow Graph (ICFG)

The ICFG is the highest abstraction used in the IR, created during ICFG reconstruction (see Section 5.4.5). It contains a set of functions and an entry function — the IR representation of the binary function to be rewritten. All other functions are either called directly or indirectly by the entry function. In the simplest case, the ICFG contains only a single function and represents a CFG.

5.3.5 Functions

Each function contains a set of superblocks and an entry superblock – the first superblock that is executen when a function is called. The entry superblock either directly or indirectly references all other superblocks via branch instructions; they are also detected and decoded during ICFG reconstruction (see Section 5.4.5).

A function can have incoming edges from functions (a function calls this function) or outgoing edges to other functions (this function calls a function). These *call edges*, for example, allow to detect recursion (e.g., for inlining functions) and unused functions (e.g., no incoming edges from other functions).

Especially for register liveness analysis (see Section 6.3), it is helpful to have fast access to all superblocks containing a return instruction. Therefore, functions can have *return edges* to contained superblocks that contain return instructions.

5.3.6 Superblocks

The IR represents sequences of consecutive instructions using superblocks (see Section 2.4) instead of basic blocks (see Section 2.3). Using superblocks has the following advantages:

- Larger blocks are used, therefore minimizing the number of blocks overall, saving space and time during analyses.
- Simple single-superblock loop unrolling (see Section 7.2) can be applied, as more types of loops can be represented using a single superblock compared to basic blocks.
- The IR can represent the layout of the original binary code more closely. No artificial branch instructions have to be inserted, to be optimized out again when regenerating binary code.

Instead of connecting two superblocks using branch instructions, they can be linked via a *chain*. Code generation places chained superblocks sequentially into the rewritten binary code, according to their position in the chain, allowing execution to fall through from one block to the next directly. Figure 5.3 shows an example of two chained superblocks. If the conditional self-branch in the second superblock of this example can be removed, both chained superblocks can be merged, resulting in a single superblock.

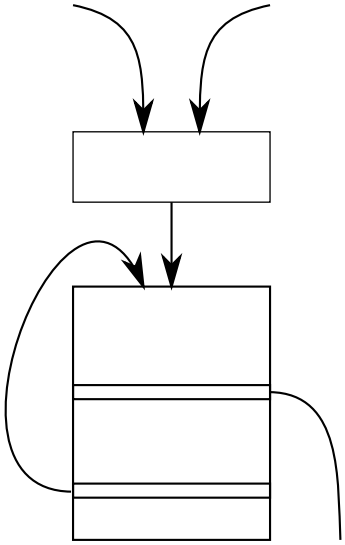


Figure 5.3: *Example of chained superblocks.*

Each superblock can have incoming edges from superblocks (another superblock branches to this superblock) and outgoing edges to other superblocks (this superblock branches to another superblock). These *branch edges* represent the edges of a CFG within each function. A superblock cannot branch to superblocks contained in other functions.

5.3.7 Registers

The IR uses architecture-specific registers, which have mostly the same name and the same meaning as the ones defined by the underlying architecture. Exceptions exist for registers that can never be explicit operands of instructions, such as the `RFLAGS` register on x86-64. Instruction encoders and decoders never have to handle such registers; IRs model them for analysis purposes only. Keeping a one-to-one mapping for explicit register

identifiers allows to efficiently translate register identifiers used in machine instructions to the identifiers defined in the IR and back. The IR does not model any additional virtual registers.

Modeling Flags

If instructions only implicitly use a register, and the individual bits of that register have dedicated meanings (i.e., flags), it makes sense to model these bits as separate registers, instead of a single, combined register with a width of multiple bits. This is helpful for analyses that do not track registers on bit granularity for efficiency reasons. If it is possible that under some condition some flags have a defined value while other flags are undefined, the analysis would have to set the whole register to undefined, resulting in an imprecise analysis.

On x86-64, this is the case with the `RFLAGS` register. In the IR, the six status flags of the `RFLAGS` register are modeled using individual, single-bit registers. The IR does not represent the remaining flags contained in the `RFLAGS` register because no IR instruction accesses them yet.

5.3.8 Instructions

As discssued, the IR supports modeled and unmodeled machine instructions. This section focuses on modeled machine instructions.

Instructions in the IR can have explicit and implicit operands. Instructions are specified using an opcode and explicit operands (e.g., a register identifier). Conditional instructions have predicates. The following design concepts were applied to define the opcodes for x86-64:

1. **The opcode defines the operands**: A specific opcode completely defines the order, the number, and the types of explicit and implicit operands, allowing to encode, decode, modify, and analyze instructions efficiently. Therefore, multiple IR opcodes might be mapped to a single machine instruction on x86-64. For example, "`ADDSD xmm1, xmm2/m64`" is split into the IR opcodes `ADDSDrr` and `ADDSDrm`, to make the type of the second operand well defined.

2. **The opcode defines the operand size**: Machine instructions that support different operand sizes have to be modeled using different opcodes, even if the operand size could be derived from the explicit operands in the IR (e.g., a 32-bit or a 64-bit register identifier is specified). For example, while the x86-64 machine instruction "`PUSH r64`" is mapped to the IR opcode `PUSH64r`, "`PUSH r16`" is mapped to the IR opcode `PUSH16r`.

3. **The address size is usually implicit**: Machine instructions that support different address sizes have to be modeled using different opcodes only if the address size cannot be derived from the explicit operands in the IR. For example, a memory address specified as an explicit operand in the IR can use either 32-bit register identifiers (e.g., `EAX`) or 64-bit register identifiers (e.g., `RAX`). The types of the

specified register identifiers determine the address size. Therefore, modeling the address size as part of the opcode is only necessary if implicit memory addresses are used. One example of an instruction that does not allow to derive the address size from the explicit operands is the "`MOVS m64, m64`" machine instruction on x86-64; both memory operands are implicit. The IR does not model this instruction yet. Possible IR opcodes would be `MOVS64_32` and `MOVS64_64` for 32-bit and 64-bit address sizes, respectively.

A full list of modeled instructions can be found in Appendix A.

Modeling Similar Instructions

Sometimes, multiple machine instructions allow specifying the same operation. Exposing these various flavors of machine instructions in the IR would not provide a real benefit; it would instead result in a significant number of similar, but slightly different opcodes.

One example is the `ADD` instruction on x86-64. To encode "ADD 2 to RAX", either "`ADD RAX, imm32`", "`ADD r/m64, imm32`" or "`ADD r/m64, imm8`" can be used. All variants have the same effect when used for this example; the only differences are the size of the immediate and whether the `RAX` register is an implicit operand. The encoder can handle these details by trying to compress immediates to smaller sizes or by selecting machine instruction variants with implicit registers. In the IR, all three machine instruction variants from this example are represented using the two opcodes `ADD64ri` and `ADD64mi`.

Instruction Operands

There is a difference between the *dynamic values* processed by an instruction at runtime and the way the location or the formation of these values is specified before runtime; instruction operands specify the latter. *Explicit operands* are similarly to the operands encoded into a machine instruction. *Implicit operands* are glued to an opocde and cannot be changed. *Predicates* are implicit operands that are never used to write dynamic values; however, the IR handles them separately as they have a dedicated meaning.

Explicit Operands The name of an opcode indicates the order and the types of all explicit operands an opcode expects. Using this scheme allows distinguishing different variants of similar opcodes. For example, the instruction defined by the IR opcode `ADD64mr` adds register content to memory content; `ADD64rm` adds memory content to register content. Table 5.1 lists all identifiers used in opcode names to indicate explicit operands.

Identifier	Description	Example (x86-64)
r	Register operand	`JMPr` branches to the address in the register.
m	Memory operand	`JMPm` branches to the address in the memory location.
a	Memory address	`JMPa` branches to the address.
i	Immediate	`PUSH16i` pushes a 16-bit immediate to the stack.

Table 5.1: *Identifiers used to indicate explicit operands in IR opcode names.*

The opcode name does not indicate all details about explicit operands (e.g., expected register types or access modes).

A register operand is specified via a register identifier. An immediate is specified via the value of the immediate. A memory operand is specified via a memory address. With a focus on x86-64, the following types of memory addresses can be specified:

- **Direct**: the absolute 64-bit address is given directly. No address calculation at runtime is necessary.
- **SIB**: the scheme used to encode memory addresses in most x86-64 machine instructions. A base register identifier, a scale (immediate), an index register identifer and a 32-bit signed displacement (immediate) are specified. The `None` register identifier can be used for the base and index register to ignore them. At runtime, the address is calculated using the actual register contents.

The encoder converts between modes if necessary and possible. For example, if a machine instruction can only encode `SIB` addresses, the encoder tries to convert `Direct` addresses to `SIB` addresses; as this implies shrinking a 64-bit value to a 32-bit signed value, this might not always be possible. Optimization passes are expected to create and fixup memory addresses so that they can be encoded properly. Fixing up memory addresses is especially relevant when it comes to RIP-relative addressing on x86-64. Section 5.4.8 discusses this topic in more detail.

Implicit Operands Opcodes in the IR can define implicit operands. Without support for implicit operands, all machine instructions relying on implicit operands would have to be split up into instructions only making use of explicit operands. This is often not possible without introducing artificial instructions not part of the ISA of the underlying architecture, causing additional effort for machine code generation.

Predicates The IR supports conditional instructions via simple predicates. A predicate is evaluated before an instruction is executed and decides whether the actual operation is performed.

Control Flow Types and Edges

Each opcode in the IR has a control flow type; this is especially helpful to handle control flow instructions in a generic way. The control flow type of an instruction is derived from the control flow type of the opcode. Table 5.2 lists the four defined types.

Type	Description	Example (x86-64)
Branch	Branch within function	`JMP`, `Jcc`
Call	Function call	`CALL`
Return	Return from function	`RET`
None	No control flow change	`ADD`, `MOV`

Table 5.2: *Defined control flow types in the IR.*

In the IR, instructions cannot branch to superblocks contained in other functions. The ICFG reconstruction would create duplicate superblocks in two functions. For example, tail calls in the original binary code would look like ordinary branches within a function.

The control flow type of an instruction and the underlying opcode might, in general, not match; for example, when architectures use the same machine instruction for branches and function calls. As the focus is on x86-64 for now, this topic is left for future work. Instructions with a control flow type of `None` never have to be processed during ICFG reconstruction.

As control flow instructions are the origin of branch, call, and return edges, each instruction can store references to such edges. A branch instruction without a branch edge indicates branching into unknown code. The same applies to function calls.

Control flow instructions can have predicates. For example, a branch instruction with a predicate represents a conditional branch instruction. Predicates are not used with other control flow instructions yet.

Original Instruction Information

Instructions can store a reference to the original binary code, along with the length of the original machine instruction. This allows supporting unmodeled machine instructions in the IR and reusing original binary code under certain conditions. More details can be found in Section 5.4.6.

5.4 Rewriting System

The discussed rewriting approach (see Section 5.1) and the described IR (see Section 5.3) were implemented in a prototype rewriting system called *Drob (Dynamic Rewriter and Optimizer of Binary code)*. Drob is written in C and C++; it is publicly available[1] under the GNU Lesser General Public License, Version 3. This section gives an introduction to Drob.

5.4.1 System Overview

Figure 5.4 shows the architecture of Drob. Given a binary function and a configuration, a rewritten binary function is created. The two main data structures used in Drob are the IR (see Section 5.3) and the binary pool, which stores the rewritten binary code and constants.

Drob uses the generic pass infrastructure to perform a set of operations on the IR, such as reconstructing the ICFG or generating binary code. The generic backend contains architecture independent functionality. The architecture-specific backend is the extension of the generic backend, responsible for all architecture-specific functionality when it comes to encoding, decoding, and emulating instructions.

[1] `https://github.com/davidhildenbrand/drob`, accessed 21-May-2019

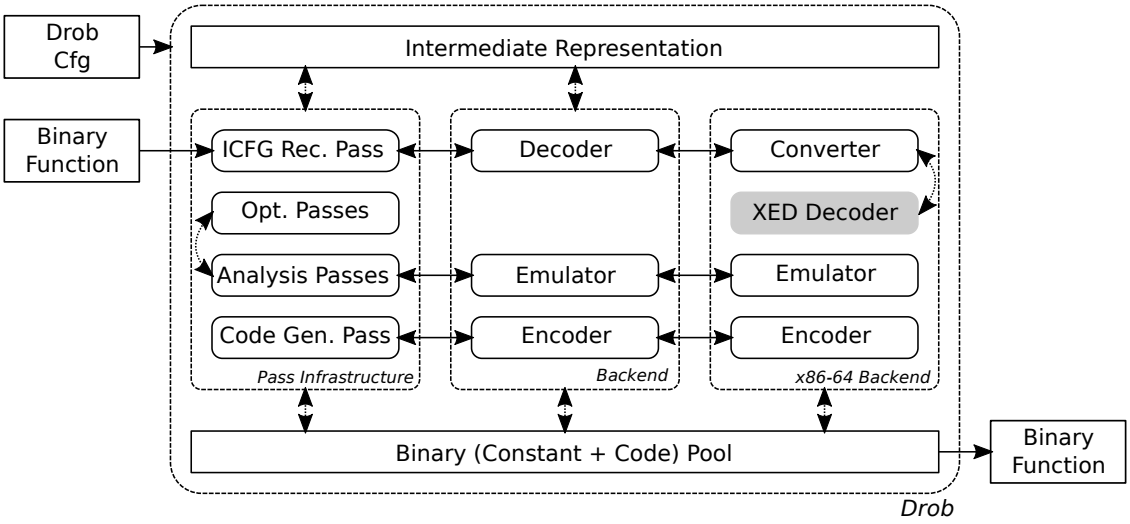


Figure 5.4: *Overview of the prototype rewriting system Drob. This figure does not show all architecture-specific backend parts.*

5.4.2 Application Programming Interface

Drob is implemented as a libary and provides a simple C interface for applications linking to it. Listing 5.1 shows an example usage of the API provided by Drob.

The rewriter configuration (Line 1) specifies all rewriter options except the binary function to be rewritten. A configuration can be reused to rewrite multiple binary functions. When a new configuration is created, the signature of the original binary function has to be specified (Line 11). The first parameter of that signature specification always corresponds to the return type of the binary function. Pointers do not have element types; the rewriter treats all pointers like `void` pointers. All configuration options are initialized to default values when creating a new configuration.

In the rewriter configuration, it is also possible to specify runtime information and other rewriting options, like error handling (Line 17). In the given example, no further runtime information about the second parameter is specified.

The return value of the rewriting request (Line 20) is a function pointer to a binary function. If rewriting fails, depending on the configuration, this might be a `NULL` pointer or even the original binary function. Therefore, a caller of the rewritten function (Line 22) still has to specify all parameters, including the ones specified via the rewriter configuration to be fixed. The rewritten binary function has the same signature as the original binary function, which is also why the original binary function can be returned in case rewriting fails.

Of course, not all possible function signatures valid in C can be expressed using this API. Only basic C data types (e.g., `int`, `long long`, or `__int128_t`) are supported. Function signatures containing composite data types — structs and unions — can be expressed using wrapper functions that only make use of the supported data types in their function signature, constructing the composites internally.

```
drob_cfg *cfg;
drob_f func;
int ret;

/* Initialize the rewriting system once, e.g., initializing decoding tables. */
drob_setup();

/* Create a new configuration, specifying the signature of the binary function
   to be rewritten: int function(const double *, double) */
cfg = drob_cfg_new2(DROB_PARAM_TYPE_INT, DROB_PARAM_TYPE_PTR,
                    DROB_PARAM_TYPE_DOUBLE);
/* Fix the first parameter (pointer) to a specific value. */
drob_cfg_set_param_ptr(cfg, 0, some_ptr);
/* Any values read via this pointer can be considered to be constant. */
drob_cfg_set_ptr_flag(cfg, 0, DROB_PTR_FLAG_CONST);
/* In case rewriting fails, return the original binary function. */
drob_cfg_set_error_handling(cfg, DROB_ERROR_HANDLING_RETURN_ORIGINAL);

/* Try to rewrite and optimize a binary function. */
func = drob_optimize(some_function, cfg);
/* Cast the generic function pointer to the actual type to call it. */
ret = ((int (*)(const double *, double))func)(some_ptr, some_data);
/* Free/release the rewritten function. */
drob_free(func);

/* Free the configuration. */
drob_cfg_free(cfg);
/* Tear down the rewriting system, e.g., free/release rewritten functions. */
drob_teardown();
```

Listing 5.1: *Example usage of the Drob API.*

5.4.3 Binary Pool

Drob uses a binary pool to store rewritten binary code (the code pool) and constants needed by instructions (the constant pool). A single binary pool is used for all functions in the ICFG. The binary pool manages data and code in a single component because this way, it can be guaranteed that instructions in the code pool can address constants stored in the constant pool using RIP-relative addressing on x86-64. x86-64 requires both pools to be positioned relatively close next to each other in the address space, so a signed 32-bit displacement relative to the address of the instruction can be used to reach the memory location of the constant.

The binary pool allocates a single memory region for both pools. While the code pool starts at the beginning of the memory region and grows towards the end, the constant pool starts at the end of the memory region and grows towards the start. Once both pools meet, the binary pool is out of memory. Figure 5.5 shows the basic memory layout of the binary pool.

Being able to use RIP-relative addressing avoids having to find and use a spare register to hold the base address of the constant pool, which highly simplifies rewriting on x86-64.

Code Pool The code pool is used to allocate memory for machine instructions during code generation and to indicate the start of new blocks of code. New blocks of code represent possible branch or call targets. The code pool always allocates memory for

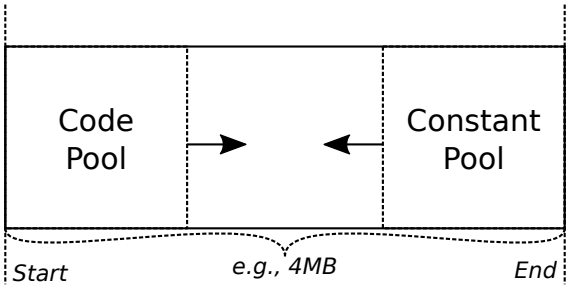


Figure 5.5: *Memory layout of the binary pool.*

new instructions sequentially, so the execution of sequentially allocated code works as expected.

In case a new block of code is indicated, the code pool allocates the next instruction from the next available address aligned to 16 bytes. The spare memory is filled with `NOP` instructions, which is necessary so the execution from the previous block can fall through to the next block, executing valid machine instructions. Aligning addresses of branch and call targets to 16 bytes is advised on x86-64 [57].

As the code generation pass processes the entry superblock in the entry function first, the first instruction in the code pool corresponds to the entry of the rewritten binary function.

Constant Pool The constant pool can store constants of different sizes. 64-bit and 128-bit constants are cached, so constants that are already in the cache can be reused. The constant pool always aligns constants in memory naturally, which is helpful if selected instructions only support naturally aligned memory access, like most Intel® SSE instructions.

5.4.4 Pass Infrastruture

The pass infrastructure is used to write and run different kinds of passes in a single framework. A pass performs a collection of operations on the IR and the binary pool. Architecture independent and architecture-specific passes are possible. The first pass Drob runs is always the ICFG reconstruction pass; the last pass is always the code generation pass, which finishes the rewriting process.

A pass can indicate that it requires specific analysis data. In case analysis data is required and not valid, the corresponding analysis pass is executed by the pass infrastructure first. Chapter 6 contains more details about provided analysis passes.

5.4.5 Decoding and ICFG Reconstruction Pass

The first step of the ICFG reconstruction is to decode the entry function. Whenever an instruction calls an undecoded function, that undecoded function is scheduled for decoding. Call edges are established after both involved functions were decoded. Return edges can be created immediately.

The ICFG reconstruction start handling a function by decoding the entry superblock into the function. Whenever an instruction branches to an undecoded superblock, that undecoded superblock is scheduled for decoding. As the IR uses superblocks, a block of code is considered to be completely decoded once an unconditional control flow instruction is detected. Branch edges are established after both involved superblocks were decoded.

Special handling is performed if a branch instruction targets an instruction that another superblock already covers, however, the target instruction is not the first instruction in that superblock. In the simple case, the containing superblock can be split before the respective instruction into two superblocks, replacing the original superblock. Both superblocks are chained so the control flow of the original superblock is maintained. This approach tries to reconstructs the original code layout, avoiding duplicate code blocks.

It may happen that the branch target instruction is not actually covered by the superblock, although the superblock spans the binary code of the original instruction. This is possible on x86-64, whereby a part of an instruction can represent another valid instruction. One example is skipping instruction prefixes by branching directly to the opcode. If this condition is detected, a new superblock is decoded, starting from the actual branch target instruction address.

The ICFG reconstruction pass only performs a static ICFG reconstruction. If targets of control flow instructions cannot be resolved without runtime information, the instructions are treated like control flow instructions branching to unknown code. Such instructions do not get any edges assigned within the ICFG.

XED Decoder and Converter On x86-64, Drob uses the open-source Intel® *XED*[2] decoder. After decoding a machine instruction, information about that instruction is available in the XED data structure. Drob converts this format to the IR instruction format. Translating explicit operands mainly consists of translating register identifiers and extracting immediates.

By using an existing decoder that supports most x86-64 instructions, even instructions not modeled in the IR can be decoded and checked for specific properties. If an instruction neither uses RIP-relative addressing nor changes the control flow, it can be represented as an unmodeled instruction in the IR

Handling LOOP and LOOPcc On x86-64, the `LOOP` and `LOOPcc` instructions are complex conditional branch instructions. These instructions decrement a register even if no branch is performed. To represent them in the IR, they can be converted to a sequence of simpler x86-64 instructions, having the same effect. The simpler x86-64 instructions can be modeled easily in the IR.

Listing 5.2 shows how the `LOOPE` instruction with a 64-bit address size can be expressed using simpler x86-64 instructions. The `LEA` instruction performs the decrement operation without modifying the `RFLAGS` register. The predicate is emulated using two conditional branch instructions and one unconditional branch instruction. Similar conversions can be done for the `LOOP` instruction (dropping the `JNZ` instruction) and the `LOOPNE` instruction

[2] `https://github.com/intelxed/xed`, accessed 21-May-2019

(replacing the JNZ instruction by a JZ instruction). The 32-bit address size variants can be handled similarly.

```
1   lea rcx, [rcx - 1]
2   jrcxz NEXT
3   jnz NEXT
4   jmp ORIGINAL_BRANCH_TARGET
5 NEXT:
```

Listing 5.2: *Converting LOOPE (64-bit address size) into simpler x86-64 instructions.*

5.4.6 Code Generation Pass

The code generation pass emits code for functions and superblocks using a Depth First Search (DFS) on the ICFG. All superblocks of a function are encoded before continuing with the next function. Therefore, functions and superblocks are processed in the order the code first needs them. Each function and each superblock is only encoded once. Using a DFS implies that the entry function is processed first; the entry superblock of each function is processed first.

Chained superblocks are processed sequentially, to avoid having to insert artificial branch instructions. Whenever the code generation pass starts to emit code for the next superblock, it instructs the code pool to align the address of the next instruction in memory properly (see Section 5.4.3).

Code Reuse The IR stores information about the original binary code of instructions. By remembering if an IR instruction has to be re-encoded, the original binary code can sometimes directly be reused, skipping the instruction encoder. Code reuse is possible if all of the following conditions are met:

- The IR instruction is based on an original machine instruction in the binary code.
- The original machine instruction does not use RIP-relative addressing (see Section 5.4.8).
- The IR instruction is not a control flow instruction whose target is a node in the ICFG, indicated by an edge.
- The IR instruction was not modified after decoding.

The decoder can directly specify if a decoded instruction has to be re-encoded, which lets the architecture backend indicate RIP-relative addressing early. By re-encoding all control flow instructions and tracking modifications of instructions in the IR, Drob realizes the reuse of original binary code in an architecture independent way — except special cases like delay slots.

Instruction Selection As the instructions modeled in the IR are based on machine instructions of the underlying architecture, instruction selection is easy and efficient. The instruction encoder for x86-64 tries to minimize the encoded instruction size by compressing immediates (e.g., use an 8-bit immediate instead of a 32-bit immediate) and using machine instruction variants with implicit operands for selected explicit operands, if possible. Also, the instruction encoder for x86-64 tries to minimize the size of memory addresses that have to be encoded, for example, by using an 8-bit displacement instead of a 32-bit displacement.

The instruction encoder does not specialize instructions; for example, an `ADD` instruction that adds an immediate of 1 to a register will not be replaced by an `INC` instruction, even if it would be valid. Optimization passes are expected to perform such modifications (see Section 7.5).

Control Flow Instruction Handling The instruction encoder implements special handling for branch instructions and function call instructions whose target is a node in the ICFG. The code address of the rewritten target node might not be known at the time the control flow instruction is requested to be encoded. So instead, space is only reserved when processing such instructions; as soon as the new binary code address of the rewritten target node is known, the actual instruction can be encoded and placed into the reserved memory location. Encoding of banch instructions and function call instructions is split into a reservation and a fixup phase.

On many architectures, there are different variants of Program Counter (PC)-relative control flow instructions, allowing to encode these instructions with fewer bytes if the target address is close in memory. As this address might not be known when reserving memory for the machine instruction, memory has to be reserved for a longer machine instruction even if unnecessary. To optimize this scenario, Drob runs the code generation pass twice.

The first run emits no binary code; it only constructs a possible layout of the whole binary code. Memory for the longest machine instruction variant is reserved. During the fixup phase, the encoder remembers if a shorter variant could have been used instead. The second run uses this information to minimize the amount of memory needed for control flow instructions. If possible, it reserves memory for shorter variants of branch instructions and function call instructions.

On x86-64, the RIP-relative `CALL` instruction can only be encoded with a 32-bit displacement, so the needed amount of memory cannot be minimized. However, most branch instructions such as `JMP`, `JZ` or `JNZ` have variants with a short (8-bit) displacement and a long (32-bit) displacement. One special case is the `JRCXZ` instruction, as it only has a variant with a short (8-bit) displacement. When rewriting code, the code layout might have changed, and the short displacement might no longer be sufficient to reach the target address. In this case, the instruction encoder emits multiple machine instructions, exploiting that the `JMP` instruction has variants with a short and a long displacement. Listing 5.3 shows a possible solution. Similar handling is performed for the `JECXZ` instruction.

```
  jrcxz CONDITIONAL_BRANCH
  jmp NEXT
CONDITIONAL_BRANCH:
  jmp ORIGINAL_BRANCH_TARGET
NEXT:
```

Listing 5.3: *Encoding JRCXZ on x86-64 with a displacement bigger than 8-bit.*

5.4.7 Memory Protection Cache

Linux provides access to information about active memory protections via the virtual file `/proc/self/maps`, which can be used by each process to query memory access permissions of memory mappings valid in its address space. For each mapping, Linux indicates whether the mapping can be read, written, and executed without triggering a signal. Parsing the list of mappings every time Drob tests a memory location is expensive. Therefore, the memory protection cache takes care of caching access information about all mappings.

Drob tries to detect if selected memory locations cannot be modified. It assumes that the content of immutable memory locations cannot change, which is helpful when decoding and optimizing. For example, the ICFG reconstruction can resolve targets of indirect branches via memory if the memory locations storing the target address are detected to be immutable. Optimization passes use the same technique to detect constants, to move them to the constant pool (see Section 7.6) and to specialize instruction to known input operands (see Section 7.5).

For example, the dynamic library loader on Linux uses the *GOT (Global Offset Table)* to redirect calls to functions in shared libraries either to the dynamic library loader or to the loaded binary code. Usually, this table is not write-protected: therefore, Drob cannot resolve indirect branches and indirect function calls via table entries. However, when *RELRO (RELocation Read-Only)* is enabled, all shared libraries are loaded when the program is loaded, and the table is write-protected. Drob can detect the table entries as constant and reconstruct the ICFG of functions in shared libraries. [40]

An application can theoretically modify memory mappings to make them writable again (e.g., using the *mprotect* system call on Linux), implying that this memory is no longer immutable. Similarly, shared mapping (*MAP_SHARED*) with other processes or files can allow third parties to modify the content even if the mapping is set as read-only. Also, some applications rely on restricted memory access permissions and signals to implement functionality like dirty-tracking. Therefore, Drob allows applications to disable the memory protection cache.

5.4.8 RIP-Relative Addressing

Various architectures support PC-relative addressing. Code and data is addressed relative to the address of the machine instruction in memory. Rewriting instructions that make use of PC-relative addressing is challenging and requires different approaches. The majority of the explained approaches should work on various architectures, but the focus is on handling x86-64.

RIP Register On x86-64, the `RIP` register holds the PC, which can be modified by an application via control flow instructions such as `JMP`, `CALL`, and `RET`. An application can extract the value of the `RIP` register via `LEA` and `CALL` instructions.

When rewriting binary code, the control flow is rewritten as well. For example, a relocated `CALL` instruction is expected to place a different address onto the stack than the original instruction, so the rewritten code works correctly. If the `LEA` instruction is used to extract the value of the RIP, it is usually used to address RIP-relative data or code, not to expect a specific value. While one could write programs that rely on the RIP register to have specific values (e.g., "from where was the function called"), this is an exotic case.

Approaches Drob can handle most control flow instructions that use RIP-relative addressing via basic ICFG reconstruction and code regeneration. For example, the ICFG reconstruction pass will determine the target address of a relative branch instruction and decode the target superblock. When generating code from the IR, the rewritten branch instruction does no longer refer to the original branch target address, but instead to the address of the rewritten superblock.

While there are control flow instructions for which this is not sufficient (e.g., an indirect branch via a memory location, addressed via RIP-relative addressing), also other instructions that use RIP-relative addressing are problematic.

Whenever the decoder processes a machine instruction and detects that a memory address is calculated using RIP-relative addressing, the decoder converts the memory address to an absolute address for the IR instruction. Drob has to find a way to encode this absolute addresses in the IR instruction into a machine instruction. Simple approaches are:

- If the rewritten machine instruction can reach the absolute address via RIP-relative addressing from its new location in memory, the encoder can use RIP-relative addressing. Although this approach does often not apply, Drob implements it in the instruction encoder.
- If the absolute address is relatively small, it can be encoded into the machine instruction directly. This case is common for binaries linked to a static address. On x86-64, this approach is possible if the address fits into a signed 32-bit value. Drob implements this approach in the instruction encoder.
- If the instruction only loads the memory address to a register, like the `LEA` instruction on x86-64, the absolute address can either be moved directly to the register via an immediate, or indirectly via the constant pool. On x86-64, the "`MOV r64, imm64`" machine instruction can load a 64-bit immediate to a register. Drob implements this approach in an optimization pass (see Section 7.5).
- If the memory locations at the absolute address are detected to be constant (see Section 5.4.7), Drob can relocate that constant into the constant pool. The new address of the constant can be reached via RIP-relative addressing from the new machine instruction location (see Section 5.4.3). Drob implements this approach in an optimization pass (see Section 7.6).

Things get more involved if the binary code accesses global variables and the described approaches fail. For example, if the "`ADD r/m64, imm32`" machine instruction on x86-64 is used to add an immediate to a global variable (via RIP-relative addressing), there is no easy way to rewrite this instruction without using an extra register. Advanced approaches would either have to identify a dead register or temporarily free up a register that can be used for addressing.

Drob does not implement advanced approaches. If a memory address cannot be encoded, rewriting is aborted. This implies that global variables of position-independent executables and shared libraries are not supported by Drob on x86-64.

6 Analyses

This chapter gives a more detailed introduction to the representation of architecture-specific registers and instructions in the IR and introduces the two implemented analyses: the register liveness analysis and the stack analysis.

Analyses make use of detailed information about registers and instructions in the IR, to allow for advanced optimizations (see Chapter 7). All analyses are limited to the entry function for simplicity.

6.1 Register Information

The IR represents each architecture-specific register generically. Modeled information consists of a register type and register hierarchy details.

Register Types Registers have a type defined by the underlying architecture (e.g., a general-purpose register) and a width (e.g., 64-bit). The IR represents both things combined via the IR register type.

Register types in the IR are architecture independent; the current types are inspired by x86-64. Examples include `Flag1` (e.g., the `OF` flag in the `RFLAGS` register), `Gprs8` (e.g., the `AH` register), `Gprs64` (e.g., the `RSP` register), and `Sse128` (e.g., the `XMM0` register).

For example, the stack analysis (see Section 6.4) uses register types to determine the location of elements and the number of elements needed to track a register in the program state.

Register Hierarchy Information In many architectures, some registers overlay other registers, either for historical reasons or to save space. For example, the `AL` and `AH` registers on x86-64 compose the `AX` register. `AX` itself overlays the lower part of the `EAX` register. The `EAX` register overlays the lower part of the `RAX` register. Figure 6.1 visualizes this example.

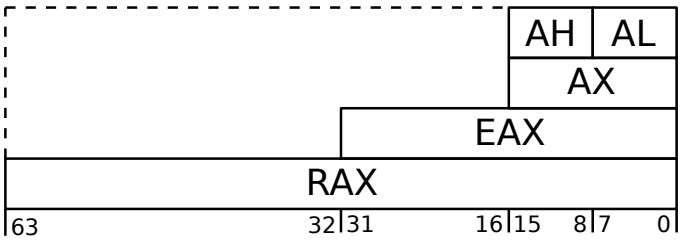


Figure 6.1: *Example of overlaying registers on x86-64.*

For example, the stack analysis (see Section 6.4) has to modify all overlaid registers when the `AX` register is modified. A stack analysis is more efficient if it can track the content of a register only at a single location, so only one location has to be updated. To implement such optimized tracking generically, the analysis needs access to register hierarchy information.

The IR allows specifying a parent register, which always corresponds to the outermost overlaid register. For example, the `AL`, `AH`, `AX`, and `EAX` registers on x86-64 specify the parent register `RAX`. Besides, each IR register with a parent IR register has an offset into the parent register. On x86-64, the four high 8-bit general-purpose registers (e.g., `AH` — the offset is one byte) need this offset.

6.2 Levels of Instruction Information

Most instructions that are part of an ISA use a set of inputs to compute a set of outputs. Inputs and outputs are described by operands. Exceptions are primarily privileged instructions, and unprivileged instructions with special side effects (e.g., the `INT` instruction on x86-64).

As explained in Section 5.3.8, there is a difference between instruction operands and the dynamic values an instruction processes at runtime. Figure 6.2 shows the possible inputs and outputs of instructions.

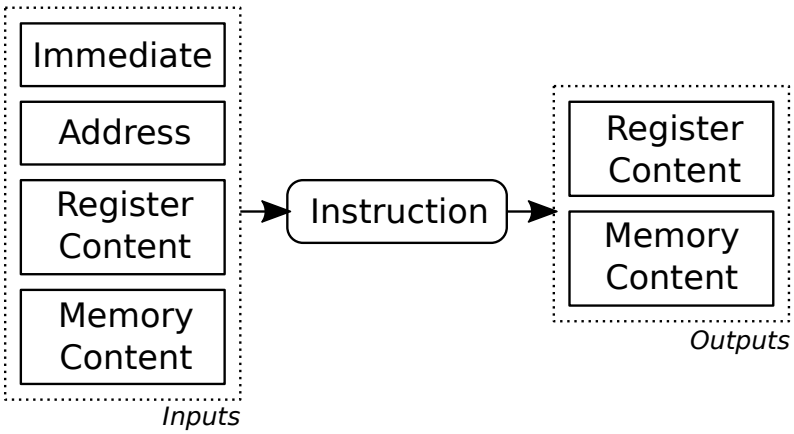


Figure 6.2: *Inputs and outputs of instructions.*

The IR stores information about each opcode. This *opcode information* has to be generic enough to hold for all valid combinations of the opcode and explicit operands. However, this information is imprecise. For example, on this level, it is unknown which register an explicit register operand uses; it is only known that the opcode expects a register identifier of a specific type. Also, other information about operands — e.g., how a register will be accessed — can be imprecise.

Once the analysis framework knows the explicit operands, it can refine opcode information to *static instruction information*. This includes *static operand information* for each operand. Explicit operands are available after decoding and translating a machine

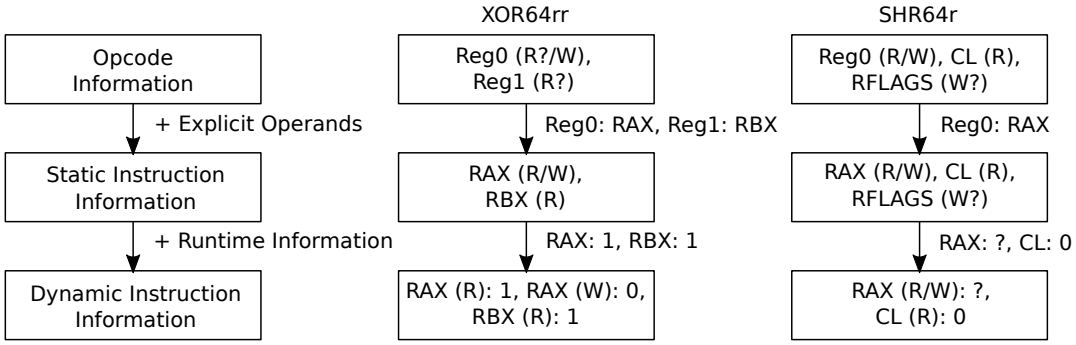


Figure 6.3: *Refining opcode information to static and dynamic instruction information.*

instruction to the IR. For example, static instruction information contains which register an instruction accesses via an explicit register operand. However, the runtime values of most operands are unknown (they are always known for immediates).

As soon as the analysis framework has runtime information about input operands available, it can refine static instruction information to *dynamic instruction information*. This includes *dynamic operand information* for each operand. The *emulator* can compute the runtime values of output operands from the runtime values of input operands. The analysis framework creates dynamic instruction information even if little or no runtime information is available; then, dynamic instruction information strongly resembles static instruction information. For example, it can already be beneficial to know that the dynamic value of an input operands will never be zero.

Figure 6.3 shows the two stages of refining information about opcodes, including two examples. Section 6.2.2 (`XOR64rr`) and Section 6.2.3 (`SHR64r`) discuss the included examples.

6.2.1 Opcode Information

The IR models information about an opcode by describing explicit operands, implicit operands, predicates, and control flow types. Section 5.3.8 introduces operand types and control flow types. Appendix A contains all opcodes modeled for x86-64 in the IR, including opcode information.

Explicit Operand Information

For each explicit operand, the expected operand type is specified, including additional information specific to each operand type.

Immediates The only information the IR models about immediate operands is the type of the expected immediate (signed or unsigned), including the size of the immediate in bits. The operand type combines both properties for simplicity. Examples include `Immediate16` and `SignedImmediate64`.

Memory Operands and Memory Addresses The IR models memory operands and memory addresses the same way, via an access mode and an access size. The access mode defines how an instruction uses a memory address. An instruction can either use the runtime value of a memory address as input directly (e.g., the `LEA` instruction on x86-64), read an input value from the computed address at runtime, or write an output value at runtime to that computed address. Conditional access can be specified. Table 6.1 lists all valid access modes for memory addresses.

Access Mode	Description
None[1]	The address is unused.
Address	The address is the dynamic value.
Read	Memory will be read.
MayRead	Memory may be read.
Write	Memory will be written.
MayWrite	Memory may be written.
ReadWrite	Memory will be read and written.
MayReadWrite	Memory may be read and will be written.
ReadMayWrite	Memory will be read and may be written.
MayReadMayWrite	Memory may be read and written.

Table 6.1: *Access modes used for memory addresses in the IR.*

The access size defines how many bytes an instruction accesses via a memory address; it is ignored if the access mode is `Address`.

Register Operands For register operands, the IR allows specifying an expected register type (see Section 6.1), an access mode, and an access type. The IR reuses the same access modes introduced for memory addresses; `Address` does not apply, and the instruction accesses a register instead of memory.

The access type specifies which parts of a register an instruction accesses. Often, instructions only access specific parts of a given register, not the whole register. For example, the "`ADDSD xmm1, xmm2/m64`" machine instruction on x86-64 expects a 128-bit XMM register identifier, however, only the lower 64-bit part of that register is accessed. The IR represents this machine instruction using the IR opcodes `ADDSDrr` and `ADDSDrm`.

The access types might differ between input and output operands. As an instruction can use the same operand for input and output at the same time, the IR models the access type for register reads and register writes separately. Examples include instruction on x86-64 that write to 32-bit general purpose registers. The "`ADD EAX, imm8`" machine instruction reads the `EAX` register, however, modifies the complete `RAX` register, writing zeroes to the upper half of the `RAX` register. This machine instruction is mapped to the IR opcode `ADD32ri`. Table 6.2 lists all valid access types for register operands.

Representing yet unmodeled x86-64 instructions or instructions of other architectures in the IR might require to add further access types. For example, the `ADDSS` machine instruction on x86-64 accesses the lower quarter of an `XMM` register — `Q0`. For performance

[1] The access mode `None` can only be the result after refining.

Access Type	Description
Full	The full register is accessed.
FullZeroParent	The full register is accessed after writing zeroes to the parent register.
H0	The lower half of the register is accessed.
H1	The upper half of the register is accessed.

Table 6.2: *Access types used for register operands in the IR.*

reasons, it might not be desirable to introduce access types for byte access (e.g., performed by the `PINSRB` machine instruction on x86-64). In this case, a relaxed access mode in combination with a coarser access type can be used — e.g., if an instruction only writes one byte of a register, leaving the remaining register untouched, this can be represented via `MayReadWrite` in combination with `Full`; analyses might become less precise.

Implicit Operand Information

The IR represents information about implicit operands similarly to information about explicit operands, however, information about implicit operands also includes the actual operand (e.g., a register identifier). Some information relevant for explicit operands does not apply to implicit operands, like the expected register type. For example, the "`SHL r/m64, CL`" machine instruction on x86-64, represented in the IR via the opcodes `SHL64m` and `SHL64r`, implicitly reads the `CL` register. As the analysis framework already knows that the instruction accesses `CL`, the IR does not model the expected register type.

Predicate Information

The IR models the predicate of conditional instructions in a format that can be evaluated efficiently. On x86-64, predicates contain simple comparisons between registers and constants. Predicates can connect at most two comparisons. Examples include the `Jcc` and the `CMOVcc` machine instructions.

A predicate in the IR is represented using a list of comparisons that can be connected by `And`($\wedge$) or `Or`($\vee$). A comparison specifies the two comparands and the comparator. Supported comparands are registers (specified via register identifiers) and immediates. Supported comparators are `Equal`($=$) and `NotEqual`($\neq$). For example, the predicate of the `JLE` opcode on x86-64 is specified as

$$(ZF = 1) \vee (SF \neq OF)$$

Appendix A includes all conditional IR instructions and defined predicates for x86-64.

6.2.2 Static Instruction Information

As described, the analysis framework refines opcode information to static instruction information. Refining includes adding explicit operands, but also changing other information about explicit and implicit operands, like the access mode.

Figure 6.3 includes an x86-64 example — the IR opcode `XOR64rr` — where a register access mode can be refined once the explicit operands are known. Implicit operands are ignored for clarity. The IR opcode `XOR64rr` corresponds to the register-register variant of the "`XOR r/m64, r64`" machine instruction on x86-64, which expects two explicit operands.

The first explicit operand is used as input and output, while the second one is used as input only. Many compilers use `XOR` instructions to set registers to zero with a minimum instruction length. For example, "`XOR RAX, RAX`" sets the `RAX` register to zero; the content of the register is irrelevant for determining the output. In contrast, "`XOR RAX, RBX`" needs the content of both registers to calculate the output. Without explicit operands, it is unknown whether the register content is relevant for an `XOR` instruction. Once the explicit operands are known, the access mode of both explicit operands can be refined, avoiding to indicate false data dependencies in analyses.

For example, the analysis framework performs this refinement for the IR opcodes `XOR64rr` and `PXOR128rr`. In the opcode information, the access mode of the first explicit operand is `MayReadWrite` and the one of the second one is `MayRead`. When refining, the access modes are converted to `ReadWrite`/`Read`, or `Write`/`None`, depending on the explicit operands.

In the example in Figure 6.3, the access modes are refined to `ReadWrite` and `Read`, because the register identifiers specified for both explicit register operands are not the same.

6.2.3 Dynamic Instruction Information

Once runtime information is available, the analysis framework tries to refine static instruction information to dynamic instruction information. It uses runtime information to create dynamic operand information, computing dynamic values of output operands from dynamic values of input operands, and refining access modes. Also, it tries to refine other instruction information. For example, sufficient runtime information might be available to evaluate the predicate of conditional instructions.

In general, the quality of available runtime information determines the quality of dynamic instruction information. In the worst case, the analysis framework has no further runtime information about input operands, implying that dynamic instruction information resembles static instruction information.

Figure 6.3 includes an x86-64 example — the IR opcode `SHR64r` — where a register access mode can be refined using available runtime information. The IR opcode `SHR64r` corresponds to the register variant of the "`SHR r/m64, CL`" machine instruction on x86-64. It has one explicit register operand and seven implicit register operands: the `CL` register and the six status flags contained in the `RFLAGS` register. The six status flags are represented using the `RFLAGS` register in this example for simplicity.

The content of the `CL` register defines if the instruction uses the six status flags contained in the `RFLAGS` as output operands or not. If the `CL` register contains the value 0, the instruction does not modify the `RFLAGS` register. However, if the `CL` register does not contain the value 0, the `RFLAGS` register is modified. Depending on available runtime information about that input operand, the access mode of the six status flags might be

refined from `MayWrite` to either `Write` or `None`.

In the example in Figure 6.3, the `CL` register contains the value 0. Although the value of the `RAX` register is unknown, the access mode of the six status flags contained in the `RFLAGS` register can be refined to `None`.

6.3 Register Liveness Analysis

The register liveness analysis identifies all registers that are *live* at each point in a function. All registers that are not live are *dead*. The content of dead registers is irrelevant for the behavior of a function. For example, a register that is written but never read is dead. The detection of dead registers allows for advanced optimizations. The optimizer can remove instruction sequences that are superfluous (see Section 7.7) and better specialize instructions (see Section 7.5).

If available, the register liveness analysis makes use of dynamic instruction information; otherwise, it uses static instruction information. For example, the register liveness analysis uses instruction information to detect if an instruction may write memory and which registers it accesses. In the context of this work, several implementation details about the register liveness analysis cannot be discussed.

The stack analysis (see Section 6.4) is responsible for generating dynamic instruction information (see Section 6.2.2). As the stack analysis needs register liveness analysis data for this process, the register liveness analysis cannot expect always to have access to dynamic instruction information. Especially during the first register liveness analysis run, only less precise static instruction information is available. This changes on successive runs, after the stack analysis has been performed.

Proving correctness, termination, and the worst-case running time of the register liveness analysis algorithm is left for future work. The algorithm has shown to work on various examples correctly and efficiently.

6.3.1 Analysis Data

The algorithm attaches register liveness analysis data to instructions and superblocks. However, it does not attach register liveness analysis data to selected instructions (See Section 6.3.2). Register liveness information is tracked in sets, whereby the set elements correspond to parts of registers worth tracking separately.

Analysis data consists of two sets: `live_out` and `live_in`. `live_out` represents all register parts that are used after an instruction or a superblock was executed. `live_in` represents all register parts that will be used before an instruction or an superblock is executed. `live_out` of unconditional branch and function return instructions is handled differently; it is the empty set, the registers that will be used after the instruction was executed are available via `live_in` of the control flow target (e.g., a superblock). Logically, the point after such an instruction in the superblock can never be reached. Consequently, `live_out` of a superblock ending with these instructions is the empty set.

6.3.2 Algorithm

The register liveness analysis only analyzes the entry function because it needs the calling convention and the signature of a function to work; this information is only currently available for the entry function in the ICFG. The algorithm propagates register liveness analysis data backward through the CFG of the entry function, along branch edges and chains of superblocks.

As the first step, the algorithm clears all existing register liveness analysis data. Starting with all superblocks that contain return instructions, the algorithm keeps detecting and processing superblocks until all superblocks have valid register liveness analysis data.

The algorithm processes a superblock by computing `live_in` analysis data from `live_out`, and attaching the analaysis data. It initializes `live_out` of a superblock either to `live_in` of the next superblock in the superblock chain or to an empty set if there is no next superblock. In case there is no next superblock, the last instruction in the superblock is an unconditional branch or function return instruction; the control flow target determines which registers are used after the instruction was executed.

Computing `live_in` of a superblock requires processing all instructions in that superblock in reverse order. `live_in` of the last analyzed instruction becomes `live_out` of the next instruction to analyze. After the algorithm analyzed the first instruction, it sets `live_in` of that superblock accordingly, and remembers if `live_in` changed.

After the algorithm processed a superblock, other affected superblocks might have to be analyzed or re-analyzed. For example, if `live_in` of a superblock changed, all superblocks branching to this superblock (via incoming branch edges) and the previous superblock in the superblock chain have to be analyzed again.

The algorithm takes care of various special cases such as evaluation orders between chained superblocks, instructions without attached liveness analysis data, and self-branches; it marks analysis data of superblocks as valid and invalid, to correctly re-analyze superblocks when necessary.

Analyzing Instructions Processing an instruction consists of calculating `live_in` of that instruction using given `live_out`, and attaching that analysis data to the instruction.

If an instruction is not a control flow instruction, never writes memory, and only performs dead register writes (`live_out` contains no registers it may write), the algorithm attaches no register liveness analysis data to the instruction. The algorithm treats such instructions as if they do not exist.

Function return instructions require special handling. The caller of a function expects certain register parts to contain return values after the call, and that the content of some registers is preserved (callee-saved) by that function during a call. As the caller of the entry function corresponds to the caller of the rewritten binary function, the algorithm computes this set using the calling convention and the signature specified for the original binary function; it sets `live_in` of the instruction to this computed set. Especially caller-saved (clobber) registers are not contained in this set. Also, the algorithm adds all other registers the function return instruction reads to `live_in`.

Function call instructions represent another special case. As the algorithm does not analyze other functions part of the ICFG, it has to treat them like unknown code. Branch

instructions that do not target superblocks in the ICFG require the same handling, as they represent branches into unknown code. The algorithm has to assume that unknown code can read any register. Therefore, it sets `live_in` to the full set.

For all other instructions, the algorithm performs the following steps:

1. Set `live_in` of the instruction to `live_out`.
2. If the instruction is always executed (unconditional, or the predicate evaluates to `true`), remove all unconditionally written register parts from `live_in`.
3. Add all register parts that the instruction reads via the predicate to `live_in`.
4. If the instruction may execute (unconditional, or the predicate does not evaluate to `false`), add all register parts it may read to `live_in`.
5. If the instruction is a branch instruction and may execute, add `live_in` of the branch target superblock to `live_in` of the branch instruction. If the algorithm has not processed the target superblock yet, `live_in` of that superblock is assumed to be the empty set. Once the algorithm analyzes the target superblock, the superblock containing the branch instruction is analyzed again.

The algorithm treats unmodeled instructions as if they read and write all registers and memory.

6.3.3 Example

One fundamental property of the algorithm is that it treats sequences of dependent instructions that do not modify memory and only write to dead registers as if they do not exist. On the one hand, this implies that the register liveness analysis is partially incorrect until the respective instructions are removed. On the other hand, the analysis data allows for implementing some optimizations (see Section 7.7) very efficiently.

Incorrect in this context means that the analysis does not indicate registers that are written by one instruction and read by another instruction as live in the register liveness analysis data attached to other instructions in between. In the context of this work, it is expected that optimization passes remove the applicable instructions, resulting in correct analysis data.

Listing 6.1 shows a simple program that initializes the `RCX` register before a loop, increments that register during the loop, and reads it once after the loop. The register liveness analysis detects the `RCX` register as live for all instructions within the loop body. This example does not include liveness information about other registers for simplicity. Imagining that the instruction in Line 7 is optimized out, also the instructions in Line 2 and 4 can be removed because this sequence of dependent instructions results in dead register writes only.

Listing 6.2 shows the result of the register liveness analysis after removing the instruction in Line 7 from Listing 6.1. The `RCX` register is no longer live. The algorithm attached no register liveness analysis data to the instructions in Line 2 and Line 4. Optimizations can identify these instructions efficiently.

```
1   ...
2   movq rcx, 5   // live_out:{rcx}, live_in:{}
3 BACK:
4   addq rcx, 1   // live_out:{rcx}, live_in:{rcx}
5   ...           // ... does not read/write rcx, writes rflags
6   jnz BACK      // live_out:{rcx}, live_in:{rcx}
7   movq rax, rcx // live_out:{},    live_in:{rcx}
8   ...           // ... does not read rcx
```

Listing 6.1: *Register liveness analysis example before removing the single consumer of a register.*

```
1   ...
2   movq rcx, 5   // no analysis data attached
3 BACK:
4   addq rcx, 1   // no analysis data attached
5   ...           // ... does not read/write rcx, writes rflags
6   jnz BACK      // live_out:{}, live_in:{}
7   ...           // ... does not read rcx
```

Listing 6.2: *Register liveness analysis example after removing the single consumer of a register.*

6.3.4 Delta Analysis

In a *delta analysis*, successive analysis runs use previously computed analysis data to improve the performance across multiple analysis runs. Old analysis data is not cleared; only modified parts (e.g., superblocks) are re-analyzed. The algorithm used does not perform a delta analysis, because the analysis data can quickly become imprecise, prohibiting optimizations. Registers can become trapped as live inside loop bodies because the old analysis data includes them as live.

A simple example where the register liveness analysis data becomes imprecise is shown in Listing 6.3. Listing 6.3 is the result of running a delta analysis on the example program from Listing 6.1, after the instruction in Line 7 was removed. When processing the `JNZ` instruction, `live_in` of the branch target superblock still contains the `RCX` register, resulting in the `RCX` register never getting detected as dead.

```
1   ...
2   movq rcx, 5   // live_out:{rcx}, live_in:{}
3 BACK:
4   addq rcx, 1   // live_out:{rcx}, live_in:{rcx}
5   ...           // ... does not read/write rcx, writes rflags
6   jnz BACK      // live_out:{rcx}, live_in:{rcx}
7   ...           // ... does not read rcx
```

Listing 6.3: *Imprecise delta register liveness analysis example after removing the single consumer of a register.*

6.4 Stack Analysis

In the context of this work, the stack analysis is responsible for creating dynamic instruction information (see Section 6.2.3) and for performing constant propagation. It identifies runtime information about the dynamic values of input operands to compute dynamic values of output operands with the help of an *emulator*.

The stack analysis data allows for advanced optimizations and detecting problematic rewriting situations (e.g., ROP). The algorithm used is designed for application-guided dynamic binary optimization and, therefore, for efficiency. An imprecise analysis is acceptable in complicated situations to keep the analysis efficient.

> "... stack analysis aims at tracking, at each program point, the state of the program, including the CPU registers, the stack of the current function and of all of its callers and the global data ..." [39]

The stack analysis algorithm used tracks CPU registers and the stack in a data structure called *program state*; it does not track global data. Exactly one program state — the entry program state — can be attached to a superblock, representing the runtime information known about the state of the program whenever that superblock is about to be executed. The stack analysis generates and attaches dynamic instruction information to all instructions part of the CFG of the entry function that are not dead.

It is not possible to give a detailed description of the algorithm, the emulator and the data structures in the context of this work. Proving correctness, termination, and the worst-case running time of the stack analysis algorithm is left for future work. The algorithm has shown to work on various examples correctly and efficiently.

6.4.1 Dynamic Values

The stack analysis uses *dynamic values* to represent runtime information about operands. During the analysis, the emulator reads dynamic values from the program state, calculates the dynamic values of output operands, and writes the outputs to the program state. Depending on the *dynamic value type*, different data is associated with a dynamic value. Some dynamic value types support different sizes (e.g., 1-byte or a 4-byte `immediate`), others support only fixed sizes (e.g., an 8-byte `usrptr`). Table 6.3 lists all supported dynamic value types, along with associated data and the size.

Type	Associated Data	Size [bytes]	Description
`unknown`	-	variable	Defined but unknown.
`dead`	-	variable	Never written, undefined.
`tainted`	-	variable	Dangerous for stack analysis.
`immediate`	1/2/4/8/16-byte value	1/2/4/8/16	Known value.
`stackptr`	8-byte offset	8	Pointer to the stack.
`usrptr`	8-byte offset, number	8	Pointer in function signature.
`returnptr`	8-byte offset, number	8	Return pointer.

Table 6.3: *Supported dynamic value types and associated data.*

`dead` is used for undefined values (e.g., a register that was never written). `unknown` represents a defined but unknown value (e.g., a register was written, however, the actual value is unknown). The type `immediate` represent completely known values (e.g., "42").

`tainted` indicates unknown values that are problematic for the correctness of the stack analysis algorithm, when used as a pointer for write accesses. For example, `tainted` is the result when multiplying a `stackptr` by an `immediate`.

The types `usrptr`, `stackptr`, and `returnptr` represent abstracted pointers. Pointers always have a size of 8 bytes. The 8-byte offset allows representing an offset from an unknown pointer base — e.g., "`stackptr` - 8". The number used for the dynamic value types `usrptr` and `returnptr` allows distinguishing different pointer bases. For example, different pointers defined in a function signature are mapped to different `usrptr`, distinguished by the number.

`stackptr` is an abstracted pointer to the stack, usually passed on x86-64 in the `RSP` register. `returnptr` represents the return address when returning from a function call in an abstracted way; the `CALL` instruction on x86-64 places it onto the stack. `usrptr` represents a pointer defined in the function signature. Each `usrptr` corresponds to a `void` pointer. Analyses and optimizations can lookup runtime information specified by the application (e.g., minimum alignment or actual value) for these abstracted pointers.

Dynamic value types are also used in the program state to track CPU registers and the stack. The program state uses some additional, internal dynamic value types that never leave the program state.

6.4.2 Program State

The program state tracks CPU registers and the stack in *elements* of byte granularity. For example, the program state uses 8 elements to track a 64-bit register. The stack corresponds to an array of elements that can grow in both directions. Each element consists of a dynamic value type, one byte of data, and an optional number (e.g., used by `usrptr` and `returnptr`). When a new program state is created, each element is initialized to the dynamic value type *dead*.

Some dynamic value types can represent single-byte values (e.g., `immediate` and `unknown`). Larger `immediate` values can be broken down into single-byte values. For example, storing an 8-byte `immediate` value to a tracked register results in the data getting distributed according to the endianness of the underlying architecture to 8 different elements, setting the dynamic value type of all elements to `immediate`. Figure 6.4 visualizes this example.

imm. 0x77	imm. 0x66	imm. 0x55	imm. 0x44	imm. 0x33	imm. 0x22	imm. 0x11	imm. 0x00
0	1	2	3	4	5	6	7

Figure 6.4: *Example of storing the 8-byte value 0x0011223344556677 on x86-64 to a tracked register in the program state.*

The program state has to handle dynamic value types with fixed multi-byte sizes (e.g., an 8-byte `usrptr`) differently. The program state uses special *tail* markers internally to

represent such a *multi-element type*; the type of the first element — the *head* — contains the real dynamic value type, and the types of the remaining elements are tail markers. For pointers, the pointer offset is distributed similar to the value of an `immediate`. Figure 6.5 shows an example, where a `usrptr` is written to a 64-bit register.

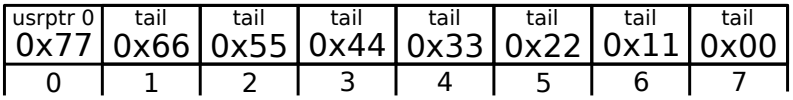


Figure 6.5: *Example of storing the 8-byte "usrptr(0) + 0x0011223344556677" on x86-64 to a tracked register in the program state.*

Using tail markers allows detecting which exact elements belong to a multi-element type, necessary when overwriting parts of a multi-element type (e.g., writing a 4-byte immediate to the lower part of a 64-bit register, containing an 8-byte `usrptr`).

6.4.3 Emulator

The emulator generates dynamic instruction information (see Section 6.2.3) for an instruction from static instruction information (see Section 6.2.2) and a program state. It modifies the program state to reflect the effects of the emulated instruction and attaches the generated dynamic instruction information to the instruction.

The architecture independent part of the emulator takes care of evaluating predicates, calculating memory addresses, reading dynamic values of input operands from the program state and global memory, and storing dynamic values of output operands to the program state. The architecture-specific part of the emulator is responsible for calculating the dynamic values of output operands from the dynamic values of input operands.

Many instructions can be emulated efficiently using corresponding machine instruction if all dynamic values of inputs have the type `immediate`. Selected instructions that can work on pointers (e.g., adding 2 to a `usrptr`, resulting in "`usrptr` + 2") require more involved emulation handlers.

Reading Dynamic Values When reading dynamic values of input operands consisting of multiple elements from a program state (e.g., reading a 64-bit register), all involved elements have to be combined into a single dynamic value; dynamic values can only have a single type (e.g., a 4-byte immediate or an 8-byte `stackptr`).

Elements can be combined easily if all dynamic value types are equal (e.g., four elements with the type `immediate`), or if all elements belonging to a multi-element type (e.g., eight elements holding a `stackptr`) are read.

If the dynamic value types are different (e.g., `immediate` and `unknown`), the emulator has to combine all involved dynamic value types. For example, combining the types `immediate` and `unknown` results in `unknown`; combining any type with `tainted` results in `tainted`.

If not all elements belonging to a multi-element type are read, the resulting type is either `unknown` or `tainted`. For example, when reading a 4-byte value from elements that contain parts of an 8-byte `stackptr`, the result has the type `tainted` — it is an unknown

piece of a stack pointer. For other partially-read multi-element types, the result has the type `unknown` (e.g., reading a single byte of an 8-byte `usrptr`).

Reading dynamic values of input operands from global memory is possible via absolute addresses (e.g., 8-byte `immediate`) and user-specified pointers (`usrptr`) with a known value. Also, Drob has to make sure that the global memory is immutable, for example, by consulting active memory protections (see Section 5.4.7). If reading from global memory is permitted, the result has the type `immediate`; otherwise, it has the type `unknown`.

Writing Dynamic Values When the emulator writes dynamic values of output operands to the program state, it may have to overwrite parts of multi-element types. For example, the emulator might store a 4-byte `immediate` to the lower part of a register that contains an 8-byte `stackptr`. In this case, the emulator has to set all elements belonging to the `stackptr` to `tainted` first. Other multi-element types have to be set to `unknown` first.

The emulator also has to take care of conditional writes. Conditional writes are possible if the emulator cannot evaluate the predicate of a conditional instruction, or if an instruction has conditional output operands. For example, if all elements of a register have the type `immediate`, and a conditional instruction writes an `usrptr` to that register, the emulator has to merge the involved elements, setting all elements to the type `unknown`. Merging of elements is also performed when merging program states (see Section 6.4.4).

Aborting Stack Tracking Under certain circumstances, the emulator can no longer guarantee that the tracked stack content is reliable; the stack analysis stops tracking the stack in the program state. Once the program state no longer tracks the stack, reading dynamic values of operands from the stack results in `tainted` and dynamic output operands targeting the stack are discarded. The stack analysis continues, but the program state only tracks registers. The generic emulator aborts tracking the stack under the following conditions:

- The emulator tries to emulate an unmodeled instruction. That instruction might modify the stack.
- An emulated instruction writes `stackptr` or `tainted` to an untracked location (e.g., non-stack memory). Another instruction could later read that value and use it to modify the stack.
- An emulated instruction uses `tainted` as address to store a dynamic value of an output operand. That instruction might modify an unknown part of the stack.
- An emulated instruction is a function call instruction. Other functions can modify any part of the stack as they are not analyzed yet.

6.4.4 Algorithm

The algorithm propagates a program state through the CFG of the entry function, along branch edges and chains of superblocks.

The algorithm attaches entry program states to all superblocks that are not dead, and dynamic instruction information to all instructions that are not dead. Consequently, superblocks without an entry program state and instructions without dynamic instruction information correspond to dead code. Optimization passes (see Section 7.4) use this fact to efficiently identify dead code that can be removed.

One important restriction of the algorithm used is that it cannot detect if stack accesses are performed via other pointers (e.g., `usrptr` or pointers stored in global variables). For example, if the caller of a binary function passes a pointer to an array on the stack as an `usrptr`, a piece of the stack could be modified by the binary function using this pointer, without the algorithm noticing it. Out-of-bounds accesses to such an array could be especially harmful. It is assumed that values residing on the stack are not accessed via the stack and other pointers at the same time by binary functions.

As the first step, the algorithm constructs the entry program state for the entry superblock using the calling convention, the function signature, and specified runtime information about parameters. Without going into details, this initializes the tracked location in the program state. For example, the initialization sets the `RSP` register on x86-64 to `stackptr`. The algorithm attaches the created program state to the entry superblock.

The algorithm starts with analyzing the entry superblock and keeps detecting and analyzing superblocks without valid stack analysis data until all superblocks have valid stack analysis data. Dead superblocks are marked to have valid stack analysis data during the analysis; however, they have no entry program state attached. The stack analysis reuses analysis data from previous stack analysis runs.

Analyzing a superblock consists of marking that superblock to have valid stack analysis data first, to handle self-branches and dead superblocks correctly. If a superblock has no entry program state attached yet, other superblocks (e.g., the previous superblock in the superblock chain) have to be analyzed first — the analysis of that superblock is complete, but it might have to be analyzed again. Otherwise, the algorithm propagates a copy of the entry program state of that superblock sequentially through the instructions of that superblock. The emulator emulates each instruction using the program state as input and output.

Whenever the emulator finishes emulating an unconditional branch instruction whose target is a superblock in the CFG, the algorithm forwards the current program state to the target superblock. For conditional branch instructions, the handling depends on the evaluated predicate. If the predicate evaluates to `false`, the algorithm does not forward the current program state to the target superblock. If the predicate cannot be evaluated (insufficient runtime information) or evaluates to `true`, the algorithm forwards the current program state to the target superblock.

The algorithm stops processing a superblock when it encounters a conditional branch instruction, and the predicate of this instruction evaluates to `true`. All instructions following this branch instruction in that superblock are dead. If the analysis detects no such instruction, processing of a superblock is finalized by forwarding the resulting program state to the next superblock in the superblock chain.

Forwarding and Merging of Program States Forwarding a program state to a superblock can mean two things. If a superblock has no entry program state attached yet, the algorithm copies the program state, attaches the copy to the superblock, and marks that superblock to have invalid stack analysis data — it has to be analyzed.

If a superblock already has an entry program state attached, the algorithm merges the new program state into the attached program state. If the merging process detects a change in the attached program state, it marks the superblock to have invalid stack analysis data — it has to be re-analyzed.

Merging of program states consists of processing each tracked location, trying to merge the elements from both program states. The resulting element generalizes both elements. Some examples for merging two elements are:

- `immediate` and `immediate` results in `immediate` if the data values match, otherwise the result is `unknown`.
- `stackptr` and `stackptr` result in `stackptr` if the pointer offsets match, otherwise the result is `tainted`.
- `usrptr` and `usrptr` result in `usrptr` if the pointer offsets and the numbers match, otherwise the result is `unknown`.
- `tainted` merged with anything results in `tainted`.
- `stackptr` merged with anything except `stackptr` (e.g., `immediate`) results in `tainted`.
- `unknown` merged with anything except `tainted` or `stackptr` results in `unknown`.

The algorithm detects a change in the attached program state if it has to modify any element.

Avoiding to Propagate Dead Registers The stack analysis algorithm uses register liveness analysis data (see Section 6.3) attached to instructions to avoid propagating register content of dead registers via the program state; this results in general performance improvements of the stack analysis.

After emulating an instruction, the algorithm sets all accessed register in the program state to `dead` that are dead according to register liveness information. For example, when the stack analysis processes an instruction that is the last one to read a register, the algorithm sets that register in the program state to `dead`.

If an instruction has no register liveness analysis data attached, the algorithm does not process it and does not attach dynamic instruction information to it. The stack analysis treats such instructions as if they do not exist, similarly to the register liveness analysis (see Section 6.3).

6.4.5 Examples

Listing 6.4 shows a simple x86-64 program that moves a value from the `RCX` register to a stack fragment. The `RCX` register is known to contain the value 5, e.g., the application

specified that value to be known at runtime. The program then moves the value 3 to the RAX register and adds that register content to the previous stack fragment. As the last step, the program moves the content of that stack fragment back to the RAX register. The stack analysis propagates the constants via the registers and the stack. Therefore, the value of the RAX register in the program state after the last instruction is 8.

```
1  ...                  // rsp is "stackptr", rcx is "5" */
2  movq [rsp - 8], rcx  // [stackptr - 8] is "5", [stackptr - 7] is "0" ...
3  movq rax, 3          // rax is "3"  */
4  addq [rsp - 8], rax  // [stackptr - 8] is "8", [stackptr - 7] is "0" ...
5  ...                  // [stackptr - 8] ... [stackptr - 1] not modified
6  movq rax, [rsp - 8]  // rax is "8"
```

Listing 6.4: *Stack analysis example where constants are propagated via registers and the stack.*

Listing 6.5 shows an example x86-64 program, where the stack analysis has to set a register to tainted because it could contain either a pointer to the stack or something unknown. The algorithm cannot evaluate the predicate of the CMOVZQ instruction, as the content of the RFLAGS register is unknown. As the RDI register contains unknown, the result of the TEST instruction is also unknown. The emulator has to set the RDI register to tainted, because it has to merge unknown from RDI with "stackptr - 8" from RSI when emulating the conditional write.

The IR does not model the CMOVZQ instruction yet; that instruction is used in this example for simplicity. Such scenarios are also possible with conditional branch instructions; the algorithm has to generalize elements when merging program states.

```
1  ...                  // rsp is "stackptr", rdi is "unknown"
2  lea rsi, [rsp - 8]   // rsi is "stackptr - 8"
3  testq rdi, rdi       // rflags are "unknown"
4  cmovzq rdi, rsi      // rdi is "tainted"
```

Listing 6.5: *Stack analysis example where* `tainted` *has to be used when emulating conditional writes.*

6.4.6 Delta Analysis

The stack analysis algorithm performs a *delta analysis*, whereby successive analysis runs use previously computed analysis data to improve the performance across multiple analysis runs. The delta stack analysis re-analyzes all superblocks that were modified since the last stack analysis run, propagating changed program states through the CFG.

Whenever optimizations modify instructions, superblocks, or functions, the binary rewriter marks the stack analysis data as invalid on all affected levels in the ICFG. For example, removing an instruction includes marking the containing superblock, the containing function, and the ICFG to have invalid stack analysis data. This way, the algorithm can identify and re-analyze modified parts efficiently.

Stack analysis data might get imprecise during the delta stack analysis. However, it should never get incorrect. Only certain modifications of the ICFG can result in an

imprecise delta stack analysis. For example, deleting dead instructions has no such effect. Identifying which modifications require an entirely new stack analysis is left for future work.

7 Optimizations

This chapter introduces the different optimizations implemented in Drob. Most optimizations require analysis data; Chapter 6 discusses the analyses. The implemented optimizations highlight which binary code modifications the optimized IR along with specialized analyses enable in Drob.

7.1 Execution Order

Drob implements six different optimization passes; it executes them in the following order:

1. **Simple loop unrolling**: unroll loops consisting of a single superblock.

2. **Block layout optimization**: merge or chain superblocks. Especially, merge unrolled loop iterations if possible.

3. **Dead code elimination**: remove dead code, including unnecessarily unrolled loop iterations. Also, convert conditional branch instructions into unconditional branch instructions where possible.

4. **Block layout optimization**: merge or chain superblocks

5. **Instruction specialization**: replace instructions by specialized instructions, for example, by encoding runtime information about dynamic values of operands into instruction.

6. **Memory operand address optimization**: minimize the number of registers needed for the memory address calculation in explicit operands, for example, by relocating detected constants to the constant pool.

7. **Dead register write elimination**: remove instructions that only result in dead register writes.

8. **Block layout optimization**: merge or chain superblocks.

Drob runs the "block layout optimization" pass multiple times, whenever there is a chance to reduce the number of superblocks and the number of branch instructions; in general, this speeds up analyses and optimizations.

7.2 Simple Loop Unrolling

The "simple loop unrolling" pass handles all superblocks that represent simple loops — single-superblock loops. Detecting and unrolling these loops requires no analysis data. The implemented approach performs some preparations on identified superblocks, to unroll each loop exactly ten times efficiently. The preparations for one superblock consist of:

1. Finding the last self-branch in that superblock. If that branch instruction is not the last instruction in that superblock, that superblock is split after the branch instruction, chaining both resulting superblocks. The superblock containing the self-branches after the split represents the loop.

2. Removing the superblock chain to the next superblock. If there is a next superblock in the superblock chain (especially after step 1), both superblocks are unchained; the chain is replaced by an unconditional branch instruction.

These preparations allow unrolling one iteration of a loop by copying the superblock representing the loop and adjusting the branch edges of the original superblock. This approach places the copy of a superblock logically after the original superblock. One unrolling step consists of:

1. Copying the original superblock representing the loop, correctly wiring up all outgoing edges of the copy. Self-branches in the original superblock remain self-branches in the copy.

2. Modifying all self-branch edges of the original superblock to target the copy.

After each step, the copy corresponds to the loop and is processed next. Repeating this process ten times results in ten unrolled loop iterations. Figure 7.1 illustrates how this approach unrolls one iteration of a loop.

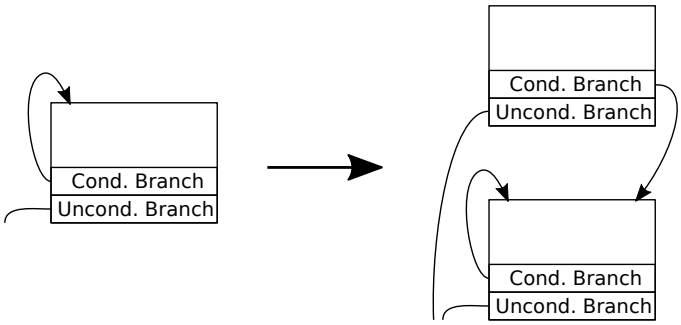


Figure 7.1: *Unrolling one iteration of a single-superblock loop.*

The implemented approach tries to perform one optimization after unrolling one iteration. If the last branch instruction in the original superblock does not target the copied superblock, the block layout optimization pass (see Section 7.3) cannot merge the superblocks representing the unrolled iterations. In this case, the second last instruction is guaranteed to be a conditional branch instruction targeting the copied superblock.

This optimization requires that there exists a conditional branch instruction with the inverted branch condition of that conditional branch instruction. For example, while the `JNB` instruction has the inverted branch condition of the `JB` instruction on x86-64, there is no such instruction for `JRCXZ`. If such an inverted conditional branch instruction exists, this optimization consist of the following steps:

1. Replacing the last instruction in the superblock — the unconditional branch instruction — by the inverted conditional branch instruction, keeping the branch edge.
2. Dropping the second last instruction — the original conditional branch instruction.
3. Chaining the original and the copied superblocks.

Figure 7.2 illustrates this optimization. This reordering allows merging all unrolled loop iterations (ten superblocks) into a single superblock, resulting in more efficient rewritten code and a faster rewriting time. In case an inverted conditional branch instruction does not exist, analyses and optimizations have to process separate superblocks, and the unconditional branch instruction in each superblock cannot be removed.

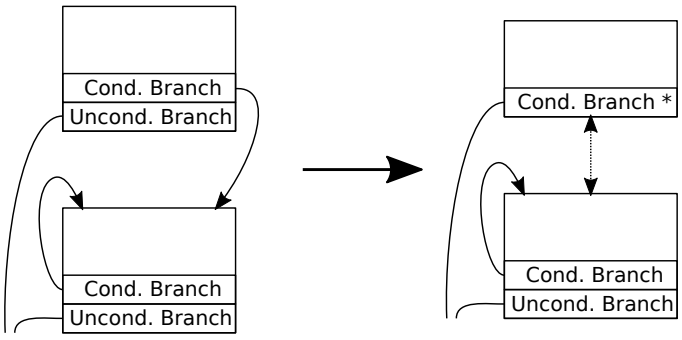


Figure 7.2: *Optimizing branches of unrolled single-superblock loop iterations.*

7.3 Block Layout Optimization

The "block layout optimization" pass consists of two simple optimizations, which reduce the number of branch instructions and superblocks. Both optimizations require no analysis data. The two optimizations are:

1. Chaining of superblocks. If the last instruction in a superblock is an unconditional branch instruction, the branch target is a superblock in the ICFG, and that superblock has no previous block in the superblock chain, the unconditional branch instruction is removed and both superblocks are chained instead.
2. Merging of superblocks. Two chained superblocks are merged if the next superblock of both superblocks in the superblock chain has no incoming edges (i.e., it is not a branch target).

This pass does not perform other optimizations (e.g., removing superblocks containing no instructions) yet.

7.4 Dead Code Elimination

The "dead code elimination pass" removes instructions and superblocks that do not affect the runtime behavior of the entry function in the ICFG. It requires valid stack analysis data, for example, because it exploits that the stack analysis (see Section 6.4) does not attach entry program states to dead superblocks and dynamic instruction information to dead instructions. All instructions without dynamic instruction information and all superblocks without an entry program state can be identified and removed efficiently.

Besides, this pass removes conditional instructions whose predicate evaluates to `false` and converts all conditional branch instructions whose predicate evaluates to `true` into unconditional branch instructions. Dynamic instruction information (see Section 6.2.3) contains runtime information about the predicate.

Similarly to the "dead register write elimination" pass (see Section 7.7), this pass implicitly removes sequences of dependent instructions that only write to dead registers — the stack analysis does not attach dynamic instruction information to instructions without register liveness analysis data (see Section 6.3).

7.5 Instruction Specialization

The "instruction specialization" pass tries to replace instructions by specialized instructions, guaranteeing an unmodified runtime behavior of the entry function in the ICFG. It encodes runtime information about dynamic values of operands into instructions, delegating most work to the architecture-specific backend. This pass only considers one instruction at a time; it does not perform multi-instruction optimizations. This section does not discuss the details of the specializations performed.

Specializing instructions requires valid stack analysis data and valid register liveness analysis data. Specializations can modify the opcode and the explicit operands of an instruction, using dynamic instruction information and register liveness analysis data to detect valid modifications. For example, a specialization can encode a known dynamic input value statically into an instruction by selecting an opcode that expects an immediate instead of a register or memory. Also, specializations might be able to replace computations by moving the pre-computed dynamic output value directly to the destination using a `MOV` instruction. The specialization process has three different outcomes:

1. No change. There is no specialized instruction for the provided dynamic values. Either runtime information about the dynamic values of operands is insufficient, or there is no applicable opcode to use for the specialization.

2. Change. There is a specialized instruction, consisting of a new opcode and new explicit operands. Various ways exist to specialize instructions. For example, liveness analysis data can be used on x86-64 to detect if the `RFLAGS` register modified by an

instruction is a dead register write; this might allow replacing an `ADD64rr` instruction by a `MOV32ri` instruction, if the content of both registers is known.

3. Delete. The instruction has no effect. For example, the `ADD64rr` instruction on x86-64 does not affect the program state if the second register contains zero, and no instruction relies on the computed `RFLAGS` register. The specialization corresponds to removing this instruction.

If needed, specializations can move constants to the constant pool (see Section 5.4.3). Appendix A includes a summary of all performed specializations of the IR opcodes for x86-64.

7.6 Memory Operand Address Optimization

The "memory operand address optimization" pass tries to minimize the number of registers needed for the memory address calculation in explicit operands, for example, by relocating detected constants to the constant pool. It requires valid stack analysis data to get access to runtime information about the dynamic values of memory operands and memory addresses. This pass is specific to x86-64.

One optimization performed by this pass is relocating detected memory constants to the constant pool. If an instruction has an explicit memory operand that it only uses as input, and the dynamic input value is known, that value is moved to the constant pool, effectively relocating a constant in memory. The modified explicit memory operand uses a direct address to the constant in the constant pool. On x86-64, the encoder can encode this address via RIP-relative addressing (see Section 5.4.8). All registers used as part of the old explicit memory operand are no longer referenced.

Other optimizations target the memory addresses of explicit operands that are specified via the `SIB` addressing scheme on x86-64; this scheme is discussed in Section 5.3.8. The optimizations are:

1. If the runtime value of a memory operand address is known and fits into a signed 32-bit value, this optimization encodes that value directly as the displacement, removing references to the `base` and `index` registers.

2. If the runtime value of the `base` register is known, and the result of adding this value to the original displacement fits into a signed 32-bit value, this optimization encodes that calculated result as the displacement, removing the reference to the `base` register.

3. If the runtime value of the `index` register is known, and the result of multiplying this value by the `scale` and adding it to the original displacement fits into a signed 32-bit value, this optimization encodes that calculated result as the displacement, removing the reference to the `index` register.

In all cases, fewer registers are needed to form memory addresses. This pass does not perform further optimizations, especially not across instructions (e.g., encoding expressions directly via memory addresses).

7.7 Dead Register Write Elimination

The "dead register write elimination" pass identifies and removes instructions that only modify registers, have no other side effect, and no instruction relies on the values of the modified registers. For example, this optimization cannot remove control flow instructions and instructions that may modify memory.

This pass requires valid register liveness analysis data to work and exploits that the register liveness analysis (see Section 6.3) does not attach register liveness analysis data to instructions that can be removed, because they only perform dead register writes. This detection approach is efficient and can remove sequences of dependent instructions.

The "dead code elimination" pass (see Section 7.4) also performs an implicit dead register write elimination. However, in contrast to the "dead register write elimination" pass, the "dead code elimination" pass requires valid stack analysis data. It is more efficient to run the "dead register write elimination" pass in case other dead code (e.g., unreachable superblocks) was already removed.

```
1  movq rax, rdi
2  movq rcx, rax
3  addq rcx, 5
4  ret
```

Listing 7.1: *Example on x86-64 where multiple dependant instructions can be removed.*

Listing 7.1 shows a simple example of a binary function on x86-64 that could be the result of other optimizations. The binary function takes one parameter via the `RDI` register and returns the result in the `RAX` register. The instructions in Line 2 and 3 are superfluous because the instruction in Line 1 already computes the result.

The register liveness analysis correctly detects that the instruction in Line 3 results in dead register writes only and treats it as if it does not exist, attaching no register liveness analysis data. Consequently, the `RCX` register is not live; the register liveness analysis does also not attach register liveness analysis data to the instruction in Line 2. The "dead register write elimination" pass can identify and remove both instructions in a single run.

8 Evaluation

An IR is only a mean to achieve a bigger goal; in the context of application-guided dynamic binary optimization, an IR has to allow for advanced code modifications efficiently at runtime. Therefore, whether an IR is optimized for this use case can be evaluated by considering the performance of the rewriting process and the rewritten code when using a binary rewriter that implements this IR.

This chapter compares Drob — implementing the IR and the rewriting approach from Chapter 5, the analyses from Chapter 6, and the optimizations from Chapter 7 — against previous work. Appendix B describes how to compile and run the benchmark.

8.1 Benchmark

This evaluation uses a benchmark that specializes a generic 2-dimensional stencil computation at runtime to a Jacobi stencil, to compute a Jacobi approximation. For example, image processing software often lets the user configure a stencil at runtime. Therefore, the stencil is unknown before runtime, and the software has to provide a generic algorithm to perform stencil computations. By specializing that generic algorithm at runtime to a specific stencil, the performance of the overall computation can be improved.

This benchmark is the same benchmark that was used to evaluate DBrew [119] and DBrew-LLVM [35]. It represents a relevant use case for application-guided dynamic binary optimization and both existing binary rewriters are guaranteed to work with this benchmark.

One Jacobi approximation in the benchmark consists of 1000 iterations. Using a matrix size of (9×9) and 80 interlines, the resulting matrix has the dimensions (649×649) and consumes ≈3.2 MB of memory. Two such matrices are used in total, one as input and one as output. After each iteration, both matrices are swapped.

Granularities The benchmark uses two types of kernel functions that operate on different granularities to compute a Jacobi approximation:

- **Element kernel**: a function to process one matrix element in an iteration. Therefore, the benchmark executes this function $649 \cdot 649$ times during each iteration.
- **Matrix kernel**: a function to process one iteration on the whole matrix. The benchmark executes this function once for each iteration.

While the element kernel functions are simpler, the matrix kernel functions can theoretically use vectorization to speed up computations across multiple elements.

Experiments with a line kernel, which processes one matrix line in an iteration at once, resulted in similar results as with the matrix kernel, which is why this evaluation does not include the line kernel granularity for simplicity.

Datatypes The benchmark uses three approaches to provide the stencil to the kernel functions:

- **Direct**: the stencil is hardcoded. Therefore, binary rewriters have no additional runtime information available.
- **Flat**: the stencil is provided via a flat data structure to the kernel function at runtime. The flat data structure consists of multiple stencil points. Each stencil point has a separate factor; the relative location of the source element in the matrix is described via a difference in x and y coordinates. Listing 8.1 shows the data structure used. Listing 8.2 contains the definition of a Jacobi stencil in the flat data structure.
- **Grouped**: the stencil is provided via a grouped data structure to the kernel function at runtime. Similarly to the flat data structure, each stencil point locates the source element in the matrix via a difference in x and y coordinates. However, all stencil points that share the same factor are grouped. Listing 8.3 contains the data structure[1] used and Listing 8.4 shows how to define the Jacobi stencil using a grouped data structure.

Depending on the configured stencil points, mathematical optimizations can be performed. For example, all stencil points of the Jacobi stencil have the same factor. Instead of multiplying each source element with the factor and adding the results, it is much faster first to add all source elements and to multiply the result once with the shared factor; this minimizes the number of arithmetical operations.

With the flat data structure, the binary rewriter has to identify equal factors in stencil points manually to perform mathematical optimizations. With the grouped data structure, this optimization is already part of the original program. In compilers, such mathematical optimizations on floating-point values that could lead to different results are usually referred to as *fast-math*.

Binary Rewriters The benchmark rewrites all six variants (datatypes and granularities) of kernel functions to create specialized binary functions using the following binary rewriters:

- **DBrew**: specialization of the native binary code using DBrew.
- **DBrew-LLVM**: specialization of the native binary code using DBrew-LLVM, referred to as "LLVM transformation with fixation" [35].
- **Drob**: specialization of the native binary code using Drob.

In addition, **Native** corresponds to the unmodified binary code.

[1] The actual data structure in the benchmark differs: StencilPoint from the flat data structure is reused.

```
typedef struct {
        int64_t xdiff, ydiff;
        double factor;
} StencilPoint;

typedef struct {
        uint64_t points;
        StencilPoint p[];
} Stencil;
```

Listing 8.1: *Flat data structure to define a stencil in the benchmark.*

```
Stencil jacobi_flat_stencil = {
        .points = 4,
        .p = {
                { .xdiff = -1, .ydiff =  0, .factor = 0.25 },
                { .xdiff =  1, .ydiff =  0, .factor = 0.25 },
                { .xdiff =  0, .ydiff = -1, .factor = 0.25 },
                { .xdiff =  0, .ydiff =  1, .factor = 0.25 },
         },
};
```

Listing 8.2: *The Jacobi stencil defined via a flat data structure.*

```
typedef struct {
        int64_t xdiff, ydiff;
} GroupedStencilPoint;

typedef struct {
        double factor;
        uint64_t points;
        GroupedStencilPoint *p;
} GroupedStencilFactor;

 typedef struct {
        uint64_t factors;
        GroupedStencilFactor f[];
} GroupedStencil;
```

Listing 8.3: *Grouped data structure to define a stencil in the benchmark.*

```
GroupedStencilPoint points[4] = {
        { .xdiff = -1, .ydiff =  0, },
        { .xdiff =  1, .ydiff =  0, },
        { .xdiff =  0, .ydiff = -1, },
        { .xdiff =  0, .ydiff =  1, },
};

GroupedStencil jacobi_grouped_stencil = {
        .factors = 1,
        .f = {
                .factor = 0.25,
                .points = 4,
                .p = points,
        },
};
```

Listing 8.4: *The Jacobi stencil defined via a grouped data structure.*

8.2 Environment

The benchmark was executed on an enterprise server and a consumer notebook. Table 8.1 lists details about both environments.

	Server	**Notebook**
CPU	Intel® Xeon® Silver 4116	Intel® Core™ i7-6600U
L1/L2/L3 Cache	32KB/1MB/16MB	32KB/256KB/4MB
#Cores	12	2
#Memory	96GB	20GB
Linux Distribution	Ubuntu 16.04.6 LTS	Fedora 29
Kernel Version	4.4.0-142-generic	5.0.6-200
gcc/g++ Version	5.4.0[2]	8.3.1
LLVM Version	4.0.0	4.0.1

Table 8.1: *Details about the environments used to run the benchmark.*

While the two matrices used in the benchmark fit into the L3 cache on the server, only one matrix fits into the L3 cache on the notebook.

Hyper-threading was active in both environments. On the notebook, Intel® Turbo Boost was disabled, and the "performance" CPU governor was enabled. The benchmark was compiled with the compiler flags "`-mno-avx -O3 -march=x86-64`".

8.3 Measured Data

Three properties were measured to compare the different approaches against each other and the original binary code.

Runtime of the Binary Code The runtime of the (original or rewritten) binary code was measured by computing 20 Jacobi approximations and calculating the average time needed to perform one Jacobi approximation, consisting of 1000 iterations. Therefore, the average runtime of the binary code represents multiple executions of the binary function, including the time needed for wrapper code to call this function multiple times and to swap both matrices after each iteration. Using a matrix with dimensions (649×649), this corresponds to $1000 \cdot 649 \cdot 649$ executions of the element granularity function and 1000 executions of the matrix granularity function.

Rewriting Time The time it takes to rewrite the original binary code was measured by performing 100 rewrites in a row, calculating the average time needed to perform one rewrite. The rewriting time consists of creating a rewriter configuration, specifying the function signature, specifying runtime information, and rewriting the binary code. The benchmark creates a new configuration for every run.

[2]Drob was compiled with gcc/g++ v8.3.1, as it requires a more recent g++ version.

Size of the Binary Code The size of the (original or rewritten) binary code was calculated using the highest and lowest code address in memory. This simple approach works reliably because the original compiler and the binary rewriters place blocks of code sequentially into memory, filling up small holes with `NOP` instructions.

8.4 Results

Figure 8.1 shows the benchmark results measured on the notebook; Figure 8.2 contains the benchmark results measured on the server.

Both compiler versions that were used result in different binary codes for all kernel functions, even for the simplest of all functions — the element kernel with a hardcoded stencil. The generated binary codes on the server and the notebook are mostly structurally identical; only for the matrix kernel with a hardcoded stencil, different binary code is generated. The result patterns between the measurements performed on the notebook and the server are the same, which is why they will not be discussed separately in most cases.

The code generated by Drob is in all cases faster than the code produced by DBrew. In contrast to DBrew, Drob generates code that is at least as fast as the original binary code (except a minor difference in one measurement). In most cases, DBrew-LLVM is able to produce the fastest code. Drob generates faster code than DBrew-LLVM in three out of twelve performed measurements.

Not considering hardcoded stencils, the code generated by Drob is 5% to 27% faster than the code generated by DBrew. The code produced by DBrew-LLVM is 31% to 82% faster than the code produced by Drob, except for the element kernel with a grouped data structure; here, the code generated by Drob is 41% to 63% faster than the code generated by DBrew-LLVM.

Looking at the rewriting times, DBrew is 8 to 25 times faster than Drob. Drob, in return, is 21 to 40 times faster than DBrew-LLVM.

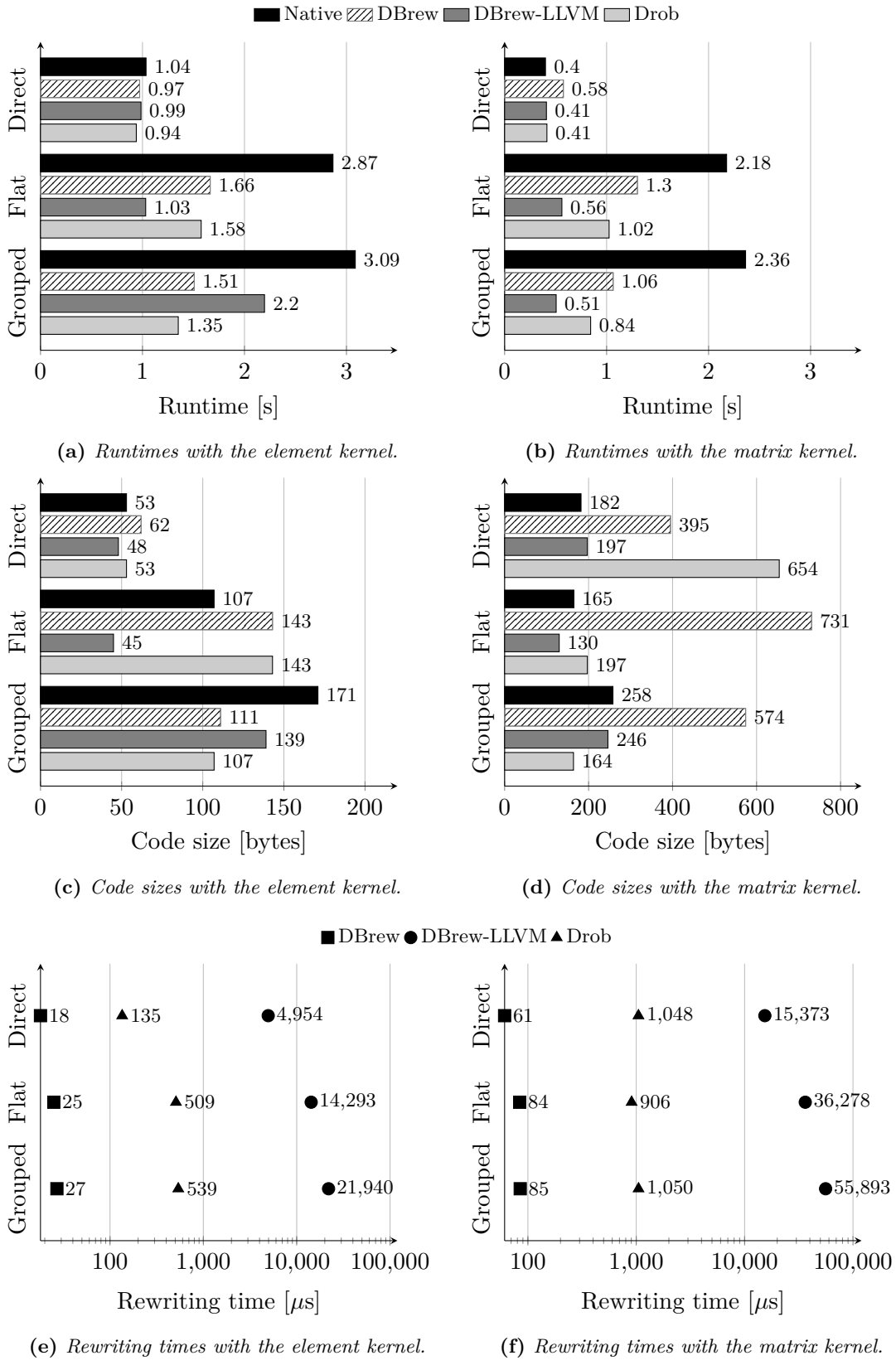
8.4.1 Hardcoded Stencil (Direct)

With a hardcoded stencil, the compiler should generate optimal binary code.

Element Kernel The older compiler on the server generates the most compact code, which is also the optimal code. The newer compiler on the notebook encodes one superfluous `LEA` instruction. In both cases, the rewritten binary code produced by Drob contains the same instructions as in the original binary code; Drob only relocates one constant to its constant pool.

For an unknown reason, relocating the original binary code and the constant results in a faster runtime of the element kernel on the server and the notebook. This phenomenon was not observed for other kernel functions.

As Drob does not perform advanced memory operand address optimizations, it cannot get rid of the `LEA` instruction contained in the original binary code on the notebook. However, the resulting performance improvement is most probably neglectable.



(a) *Runtimes with the element kernel.*

(b) *Runtimes with the matrix kernel.*

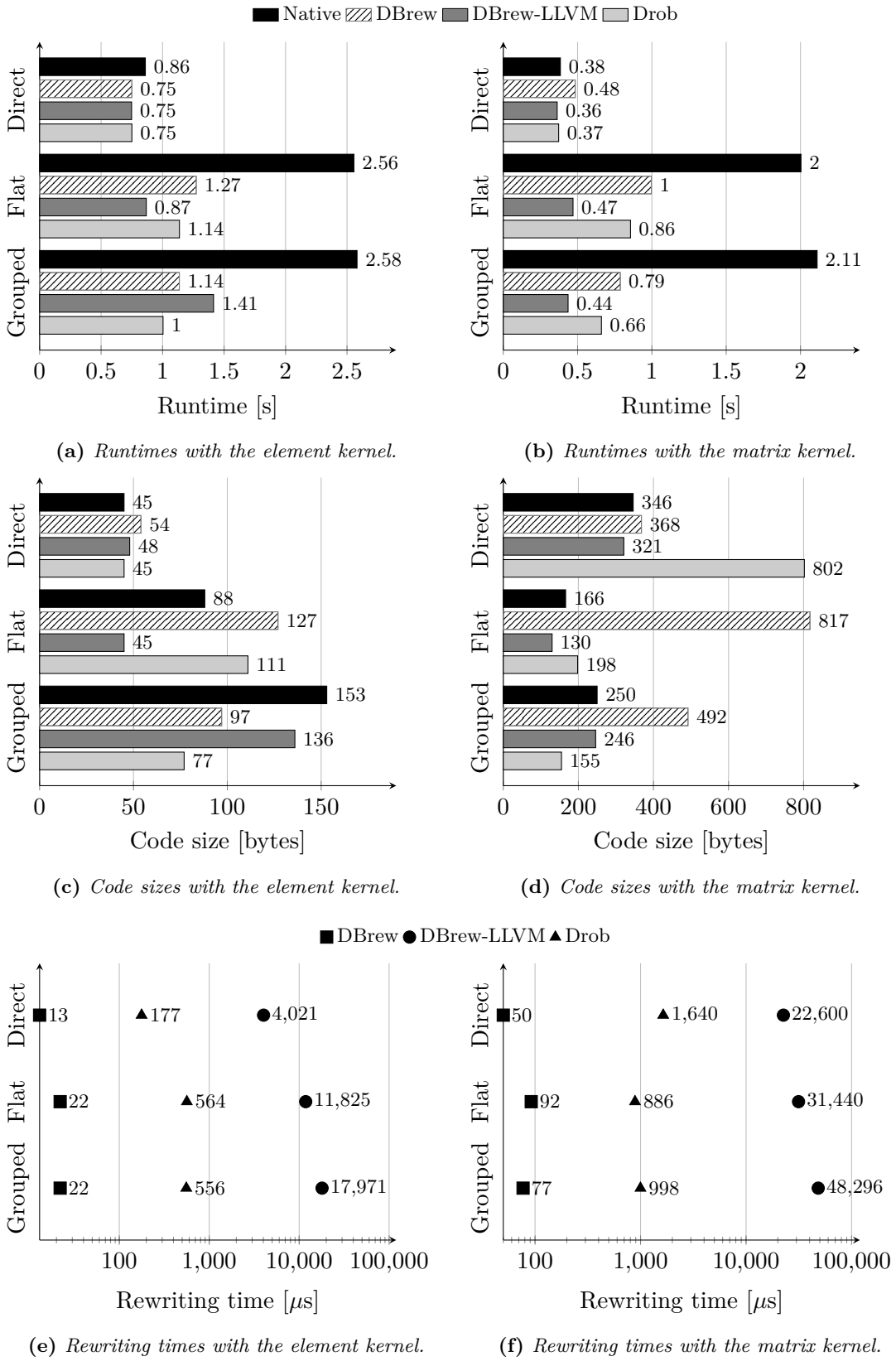
(c) *Code sizes with the element kernel.*

(d) *Code sizes with the matrix kernel.*

(e) *Rewriting times with the element kernel.*

(f) *Rewriting times with the matrix kernel.*

Figure 8.1: *Benchmark results on the notebook.*



(a) *Runtimes with the element kernel.*

(b) *Runtimes with the matrix kernel.*

(c) *Code sizes with the element kernel.*

(d) *Code sizes with the matrix kernel.*

(e) *Rewriting times with the element kernel.*

(f) *Rewriting times with the matrix kernel.*

Figure 8.2: *Benchmark results on the server.*

Matrix Kernel The compiler on the notebook generates the most compact and fastest code. On the server, DBrew-LLVM produces the most compact and fastest code; however, the performance improvement compared to the original binary code is small.

In both environments, the code generated by Drob is the largest, because Drob unrolls a long loop ten times. While Drob merges all unrolled loop iterations into one superblock without conditional branch instructions and eliminates other instructions, the code belonging to the unrolled loop still dominates the other code. As Drob has to process many instructions, the rewriting times are relatively long; partially unrolling this loop does not seem to be beneficial.

Drob also removes some `MOV` instructions by relocating constants to the constant pool and referencing the relocated constants directly from instructions. For example, Drob optimizes the code fragment in Listing 8.5 to the code fragment in Listing 8.6

```
1 movsd xmm4, [const_value]
2 ...
3 movapd xmm5, xmm4
4 ...
5 mulsd xmm0, xmm5
```

Listing 8.5: *Example 1 — original binary code.*

```
1 mulsd xmm0, [relocated_const_value]
```

Listing 8.6: *Example 1 — optimized binary code generated by Drob.*

However, the `MULSD` instruction is part of a loop; loading the value to a register once could be more efficient. While the code generated by Drob on the server is slightly faster than the original binary code, the original code on the notebook is slightly faster than the rewritten code from Drob. As the differences are minimal, this could be a measurement anomaly.

Similarly to the other rewriters, Drob does not make use of information about the minimum guaranteed alignment of pointers in optimizations yet. A possible optimization could simplify the code fragment in Listing 8.7 to the code fragment in Listing 8.8.

```
1 movupd xmm5, [rax + rdx]
2 addpd xmm0, xmm5
```

Listing 8.7: *Example 2 — current binary code.*

```
1 addpd xmm0, [rax + rdx]
```

Listing 8.8: *Example 2 — possible optimized binary code.*

8.4.2 Flat Data Structure (Flat)

The Jacobi stencil is provided at runtime to a generic kernel function via a flat data structure. Listing 8.2 contains the Jacobi stencil used.

Element Kernel The binary code generated by the two compilers consists of a loop that iterates over all stencil points.

Drob unrolls this loop completely, resulting in four unrolled loop iterations. The resulting code is a single stream of instructions, without any branch instructions. Drob is also able to remove several `LEA` and `MOV` instructions.

Drob misses many optimizations performed by DBrew-LLVM that would require register renaming and advanced memory operand address optimizations. For example, the code fragment in Listing 8.9 could be simplified to the code fragment in Listing 8.10.

```
1 mov rax, rcx
2 movsd xmm0, [rsi + 8 * rax]
```

Listing 8.9: *Example 3 — current binary code produced by Drob.*

```
1 movsd xmm0, [rsi + 8 * rcx]
```

Listing 8.10: *Example 3 — possible optimized binary code.*

Also, DBrew-LLVM performs the described mathematical optimizations that Drob does not support yet. Instead of multiplying elements with the same factor and adding the results, the binary code produced by DBrew-LLVM adds all elements first and multiplies the result with the factor once. The resulting code corresponds to the optimal code with a hardcoded stencil.

Matrix Kernel The binary code generated by the two compilers consists of three nested loops. The two outermost loops iterate over each element in the matrix; the inner loop iterates over each stencil point.

Drob unrolls the inner loop completely, similarly as with the element kernel; it misses advanced code optimizations again. For example, while DBrew-LLVM performs mathematical optimizations, Drob does not support such optimizations.

The original binary code temporarily spills the `RBX` register to the stack. The rewritten binary code in Drob no longer references the `RBX` register; however, in contrast to DBrew-LLVM, it cannot eliminate this unnecessary register spill, resulting in two superfluous instructions in the produced binary code.

8.4.3 Grouped Data Structure (Grouped)

The Jacobi stencil is provided at runtime to a generic kernel function via a grouped data structure. Listing 8.4 contains the Jacobi stencil used.

Element Kernel The binary code generated by the two compilers consists of two nested loops that iterate over each stencil point in each factor group.

Drob completely unrolls both loops, resulting in very efficient code. In contrast to the kernel functions that expect a single stencil, no mathematical optimizations are required; this is the main advantage when using the grouped data structure. Drob primarily only misses advances memory operand address optimizations, which would allow eliminating even more instructions.

DBrew-LLVM unrolls both loops similarly; however, it does not detect the stencil points as constants. Instead of relocating the stencil points, it generates code to address the original stencil points, requiring several `IMUL` and `ADD` instructions. Consequently, the code generated by Drob is faster than the code generated by DBrew-LLVM.

Matrix Kernel The binary code generated by the two compilers consists of four nested loops. The two outermost loops iterate over each element in the matrix; the two innermost loops iterate over each stencil point in each factor group.

Drob generates the most compact code of all binary rewriters. Similarly to the element kernel, Drob fully unrolls the two innermost loops and misses advanced memory operand

address optimizations. Besides, Drob is not able to remove three unnecessary register spills.

DBrew-LLVM also unrolls the two innermost loops and generates code to address the original stencil points. DBrew-LLVM removes three unnecessary register spills. In summary, the code generated by DBrew-LLVM is faster than the code produced by Drob, even though DBrew-LLVM does not treat the stencil points as constants. The reason is that the stencil points are only loaded once for the whole matrix, and not for each element.

8.5 Discussion

In the benchmark, Drob can already perform many optimizations on existing binary code, resulting in significant performance improvements compared to the original binary code and the binary code generated by DBrew. Drob can unroll simple loops, move constants to the constant pool, encode constants directly into instructions, free up many registers, and eliminate many unnecessary instructions (dead code).

Drob misses optimization opportunities because it does not implement the following optimizations yet:

- Advanced memory operand address optimizations. E.g., moving the calculation of an expression directly into the specification of a memory operand address.
- Register renaming. E.g., eliminating `MOV` instructions by renaming the registers used by instructions.
- Mathematical optimizations. E.g., reducing the number of arithmetical operations in a fast-math mode.
- Duplicate load/store elimination and dead store elimination for the stack. E.g., removing unnecessary register spills.

The simple loop unrolling approach implemented by Drob is in many cases powerful enough to optimize the code which iterates over all stencil points heavily. However, this static approach is not always beneficial. In one scenario, it resulted in larger code and longer rewriting times without improving the performance of the rewritten binary code.

In contrast to Drob and DBrew-LLVM, DBrew has the drawback that it sometimes generates code that is less efficient than the original binary code; this is especially the case with hardcoded stencils. While DBrew spends very little time rewriting binary code, the code generated using its partial evaluation approach is not as performant as the code generated by DBrew-LLVM and Drob; this is also reflected in the larger code sizes that DBrew produces.

DBrew-LLVM lacks an interface to configure memory ranges as read-only; it cannot detect the stencil points to be constant in all scenarios, resulting in less efficient code. In general, constant addresses are difficult to handle in LLVM. In contrast, Drob provides an interface to specify memory ranges to be constant, which is why it can always detect the stencil points as constants, and generate more efficient code than DBrew-LLVM in one scenario. No binary rewriter generated vectorized binary code.

Which binary rewriter is more beneficial depends on the original binary function and the number of times a rewritten binary function is reused. For example, if the matrix kernel (flat data structure) was only executed once instead of 1000 times, optimizing with Drob and DBrew would be beneficial; optimizing with DBrew-LLVM would not be beneficial.

Rewriter	**Runtime (orig.) [ms]**	**Runtime (rew.) [ms]**	**Rewriting Time [ms]**	**Saved Time [ms]**
DBrew	2.0	1.00	0,09	0.91
DBrew-LLVM	2.0	0.47	55.89	-54.36
Drob	2.0	0.86	0.88	0.26

Table 8.2: *Comparison of the overall saved time (runtime + rewriting time) when rewriting and executing the matrix kernel (flat data structure) exactly once on the server.*

Table 8.2 compares the overall saved time when rewriting and executing the matrix kernel (flat data structure) exactly once on the server. Of course, the results are different in other scenarios. For example, rewriting an element kernel function to execute it once would not be beneficial with any binary rewriter discussed in this evaluation.

As the IR in Drob allows to perform advanced optimizations of binary code at runtime efficiently, it is optimized for the use case of application-guided dynamic binary optimization. It is more powerful than representations that basically correspond to machine code (like the IR in DBrew) while allowing for more efficient analyses and modifications of binary code at runtime than LLVM IR.

9 Summary

This thesis analyzed IRs in existing binary rewriters to create an optimized IR for the use case of application-guided dynamic binary optimization. The resulting machine-level IR uses architecture-specific instructions and registers; it tries to minimize the rewriting time by allowing for an efficient translation between binary code and the IR, maintaining much work performed by the original compiler. Drob, a prototype binary rewriting system, was introduced to evaluate which analyses and optimizations are possible using this optimized IR, and how efficient the rewriting process is. Two implemented analyses — the register liveness analysis and the stack analysis — and various optimizations were described.

While Drob, including its IR, has an initial focus on x86-64 only, it was designed with retargetability in mind.

The evaluation showed that Drob can generate more performant binary code than a simple approach based on partial evaluation. However, the binary code is, in most cases, not yet as fast as the binary code optimized via LLVM. While Drob needs more time for rewriting than the simpler approach, the rewriting process is an order of magnitude faster compared to the LLVM-based approach. Thus, there is plenty of room for more analyses and optimizations at runtime to further optimize binary code.

The IR in Drob is powerful enough to allow for advanced analyses and invasive code modifications, while still focusing on efficiency. It is more powerful than simple machine-level IRs and more efficient than architecture independent high-level IRs.

9.1 Future Work

Drob is open-source and can now be used for further experiments with analyses and optimizations. As Drob is a prototype, much effort will be needed to make it stable and to model many important x86-64 instructions. Modeling new instructions in the IR, including the implementation of emulation and specialization handlers, is still time-consuming. Various code refactorings and code simplifications will make it easier to represent new instructions in the IR, and make Drob, in general, more efficient.

A critical limitation of Drob is that it only performs analyses and optimizations on the entry function, not on other functions contained in the ICFG. A straightforward approach to overcome this limitation in many situations is to conditionally inline functions into the entry function. More complicated approaches would have to detect the signature of functions in the ICFG, similarly to approaches discussed in work by Di Federico [39].

Rewriting Infrastructure Some simplifications (see Section 5.2) had to be made when it comes to general rewriting of binary code without performing invasive code modifications, as this topic is already very complicated on its own. The most important future items to look into are:

- **Reliable detection of problematic binary functions**: Drob should detect unsupported binary functions (e.g., functions that use ROP) reliably so that it can abort rewriting.

- **Dynamic ICFG reconstruction**: currently, Drob performs only a static ICFG reconstruction. Detecting the target of indirect branch instructions and indirect function call instructions would, for example, require incorporating a lightweight stack analysis into the ICFG reconstruction pass. Also, detecting jump tables and integrating them into the IR can be implemented in this context.

- **Lazy ICFG reconstruction**: it is sometimes inefficient to decode and reconstruct functions and superblocks before it is clear whether they are actually relevant. For example, the first "dead code elimination" pass might immediately remove whole superblocks again. Also, lazy decoding can be used in this context.

- **Segmentation on x86-64**: supporting segmentation is required to rewrite binary functions that use thread-local variables on Linux.

Analyses Regarding analyses, interesting future work includes performance improvements of the existing analyses, more precise analyses, and new analyses needed by additional optimizations. Especially the following items are relevant:

- **Delta register liveness analysis**: as discussed in Section 6.3.4, realizing a precise delta liveness analysis might be problematic. If possible, a precise delta register liveness analysis can result in shorter rewriting times.

- **More precise register liveness analysis**: the existing register liveness analysis approach can sometimes only use static instruction information (see Section 6.3). Different types of register liveness analyses might be needed — e.g., a static register liveness analysis executed before the stack analysis and a dynamic register liveness analysis performed after the stack analysis.

- **More precise delta stack analysis**: as discussed in Section 6.4.6, after some modifications of the binary code represented in the IR, a delta stack analysis will result in imprecise analysis data. These modifications have to be identified, to perform a completely new stack analysis in these situations instead.

- **More precise stack analysis**: conditional instructions (e.g., `Jcc` and `CMOVcc`) can result in imprecise stack analysis data. Section 6.4.5 contains one example. Splitting the control flow into separate paths (e.g., duplicating superblocks) can enable a more precise stack analysis on the separated paths. Instructions like `CMOVcc` have to be expressed using conditional branch instructions instead (e.g., replacing it by `Jcc` and `MOV` instructions).

- **Stack liveness analysis**: Drob cannot eliminate dead stores to stack locations yet.

Optimizations There are several possible optimizations that are not yet performed by Drob. The following optimizations are especially interesting:

- **Register renaming**: renaming selected registers allows getting rid of instructions that perform unnecessary register-to-register moves.
- **Conditional function inlining**: inlining function can improve the performance of the generated code; it also allows analyzing and optimizing more binary code with the current analyses that only work on the entry function of the ICFG. Inlining might not always be possible (e.g., recursion) or desired (e.g., code size).
- **Register spill elimination**: register spills can be removed once the spilled registers are no longer used in the binary code. Combining this optimization with register renaming might make it possible to remove even more register spills.
- **Advanced memory operand address optimizations**: on x86-64, some expressions (e.g., $8 \cdot \texttt{RAX}$) can be moved into the memory address specification of memory operands directly.
- **Mathematical optimizations**: various optimizations make mathematical computations more efficient. One example is $a \cdot c + b \cdot c = (a + b) \cdot c$. This requires a fast-math mode for floating-point operations.
- **Duplicate load/store elimination**: if a register or stack locations already contain the value to be loaded/stored, the respective load/store instruction might be removed.
- **Dead store elimination for the stack**: instructions that store values to stack locations that are never read might be removed.
- **Move optimizations**: instructions that produce values in specific registers or stack locations can sometimes be moved closer to the instructions that consume these values; for example, into the superblocks where the consumers reside. Then, paths that do not need these values also do not perform the computations.
- **Natural loop detection**: the implemented loop unrolling approach only considers loops consisting of a single superblock. Natural loops are easy to optimize because they only have a single entry point (the header) [7]. Detecting natural loops allows unrolling loops consisting of multiple superblocks.
- **Dynamic loop unrolling**: currently, Drob unrolls detected loops always ten times. Detecting the loop condition allows conditionally unrolling loops and developing better heuristics for the number of loop iterations to unroll.
- **Vectorization, re-vectorization, and parallelization**: there are various possible approaches to optimize detected loops and to optimize already vectorized loops.

A Modeled x86-64 Opcodes in the IR of Drob

Table A.3 contains all opcodes that Drob models in its IR for x86-64, including definitions of expected explicit operands, implicit operands, predicates, control flow types, and implemented specializations. Table A.1 contains all acronyms used in Table A.3 for explicit operands. Table A.2 contains all acronyms used in Table A.3 for predicates. For simplicity, Table A.3 does not contain all details about implicit operands, and the `RFLAGS` register is used as a representative for the six contained status flags.

Table A.1: *Acronyms used for expected explicit operands in Table A.3.*

Acronym	Operand Type	Reg. Type	Reg. Acc. (R)	Reg. Acc. (W)	Acc. Size [bytes]
r8	Register	Gprs8	Full	Full	-
r16	Register	Gprs16	Full	Full	-
r32	Register	Gprs32	Full	FullZeroParent	-
r64	Register	Gprs64	Full	Full	-
x64	Register	Sse128	H0	H0	-
x128	Register	Sse128	Full	Full	-
m	Memory Address	-	-	-	0
m8	Memory Address	-	-	-	1
m16	Memory Address	-	-	-	2
m32	Memory Address	-	-	-	4
m64	Memory Address	-	-	-	8
m128	Memory Address	-	-	-	16
i8	Immediate8	-	-		
i16	Immediate16	-	-	-	-
i32	Immediate32	-	-	-	-
i64	Immediate64	-	-	-	-
s32	SignedImmediate32	-	-	-	

Table A.2: *Acronyms used for predicates in Table A.3.*

Acronym	Predicate
B	$CF = 1$
Z	$ZF = 1$
S	$SF = 1$
O	$OF = 1$
P	$PF = 1$
NB	$CF = 0$
NZ	$ZF = 0$
NS	$SF = 0$
NO	$OF = 0$
NP	$PF = 0$
ECX0	$ECX = 0$
RCX0	$RCX = 0$
L	$SF \neq OF$
NL	$SF = OF$
BE	$CF = 1 \vee ZF = 1$
LE	$ZF = 1 \vee SF \neq OF$
NBE	$CF = 0 \wedge ZF = 0$
NLE	$ZF = 0 \wedge SF = OF$

Table A.3: *Modeled x86-64 opcodes in the IR of Drob.*

Opcode	Explicit Operands	Implicit Operands	Predicate	Control Flow Type	Specializations
ADD8rr	r8(R/W), r8(R)	RFLAGS(W)	-	-	-
ADD8rm	r8(R/W), m8(R)	RFLAGS(W)	-	-	-
ADD8ri	r8(R/W), i8	RFLAGS(W)	-	-	-
ADD8mr	m8(R/W), r8(R)	RFLAGS(W)	-	-	-
ADD8mi	m8(R/W), i8	RFLAGS(W)	-	-	-
ADD16rr	r16(R/W), r16(R)	RFLAGS(W)	-	-	-
ADD16rm	r16(R/W), m16(R)	RFLAGS(W)	-	-	-
ADD16ri	r16(R/W), i16	RFLAGS(W)	-	-	-
ADD16mr	m16(R/W), r16(R)	RFLAGS(W)	-	-	-
ADD16mi	m16(R/W), i16	RFLAGS(W)	-	-	-
ADD32rr	r32(R/W), r32(R)	RFLAGS(W)	-	-	-
ADD32rm	r32(R/W), m32(R)	RFLAGS(W)	-	-	-
ADD32ri	r32(R/W), i32	RFLAGS(W)	-	-	-
ADD32mr	m32(R/W), r32(R)	RFLAGS(W)	-	-	-
ADD32mi	m32(R/W), i32	RFLAGS(W)	-	-	-

Continued on next page.

Table A.3 – *Continued from previous page.*

ADD64rr	r64(R/W), r64(R)	RFLAGS(W)	-	-	*Delete*, ADD64rm, ADD64ri, MOV64rr, MOV64ri, XOR64rr
ADD64rm	r64(R/W), m64(R)	RFLAGS(W)	-	-	*Delete*, ADD64ri, MOV64rm, MOV64ri, XOR64rr
ADD64ri	r64(R/W), s32	RFLAGS(W)	-	-	*Delete*, MOV64ri, XOR64rr
ADD64mr	m64(R/W), r64(R)	RFLAGS(W)	-	-	*Delete*, ADD64mi, MOV64mr, MOV64mi
ADD64mi	m64(R/W), s32	RFLAGS(W)	-	-	*Delete*, MOV64mi
ADDPDrr	x128(R/W), x128(R)	-	-	-	ADDPDrm, MOVAPDrm, PXOR128rr
ADDPDrm	x128(R/W), m128(R)	-	-	-	MOVAPDrm, PXOR128rr
ADDSDrr	x64(R/W), x64(R)	-	-	-	ADDSDrm, MOVSDrm
ADDSDrm	x64(R/W), m64(R)	-	-	-	MOVSDrm
CALLr	r64(R)	RSP(R/W), [RSP - 8](W)	-	Call	-
CALLm	m64(R)	RSP(R/W), [RSP - 8](W)	-	Call	-
CALLa	m(Addr)	RSP(R/W), [RSP - 8](W)	-	Call	-
CMP8rr	r8(R), r8(R)	RFLAGS(W)	-	-	CMP8ri
CMP8rm	r8(R), m8(R)	RFLAGS(W)	-	-	CMP8ri
CMP8ri	r8(R), i8	RFLAGS(W)	-	-	-
CMP8mr	m8(R), r8(R)	RFLAGS(W)	-	-	CMP8mi
CMP8mi	m8(R), i8	RFLAGS(W)	-	-	-
CMP16rr	r16(R), r16(R)	RFLAGS(W)	-	-	CMP16ri
CMP16rm	r16(R), m16(R)	RFLAGS(W)	-	-	CMP16ri
CMP16ri	r16(R), i16	RFLAGS(W)	-	-	-
CMP16mr	m16(R), r16(R)	RFLAGS(W)	-	-	CMP16mi
CMP16mi	m16(R), i16	RFLAGS(W)	-	-	-
CMP32rr	r32(R), r32(R)	RFLAGS(W)	-	-	CMP32ri
CMP32rm	r32(R), m32(R)	RFLAGS(W)	-	-	CMP32ri
CMP32ri	r32(R), i32	RFLAGS(W)	-	-	-
CMP32mr	m32(R), r32(R)	RFLAGS(W)	-	-	CMP32mi
CMP32mi	m32(R), i32	RFLAGS(W)	-	-	-
CMP64rr	r64(R), r64(R)	RFLAGS(W)	-	-	CMP64rm, CMP64ri
CMP64rm	r64(R), m64(R)	RFLAGS(W)	-	-	CMP64ri
CMP64ri	r64(R), s32	RFLAGS(W)	-	-	-
CMP64mr	m64(R), r64(R)	RFLAGS(W)	-	-	CMP64mi
CMP64mi	m64(R), s32	RFLAGS(W)	-	-	-
JNBEa	m(Addr)	-	NBE	Branch	-
JNBa	m(Addr)	-	NB	Branch	-
JBa	m(Addr)	-	B	Branch	-
JBEa	m(Addr)	-	BE	Branch	-
JCXZ32a	m(Addr)	-	ECX0	Branch	-
JCXZ64a	m(Addr)	-	RCX0	Branch	-
JZa	m(Addr)	-	Z	Branch	-
JNLEa	m(Addr)	-	NLE	Branch	-
JNLa	m(Addr)	-	NL	Branch	-
JLa	m(Addr)	-	L	Branch	-
JLEa	m(Addr)	-	LE	Branch	-

Continued on next page.

Table A.3 – *Continued from previous page.*

JNZa	m(Addr)	-	NZ	Branch	-
JNOa	m(Addr)	-	NO	Branch	-
JNPa	m(Addr)	-	NP	Branch	-
JNSa	m(Addr)	-	NS	Branch	-
JOa	m(Addr)	-	O	Branch	-
JPa	m(Addr)	-	P	Branch	-
JSa	m(Addr)	-	S	Branch	-
JMPr	r64(R)	-	-	Branch	-
JMPm	m64(R)	-	-	Branch	-
JMPa	m(Addr)	-	-	Branch	-
LEA16ra	r16(W), m(Addr)	-	-	-	-
LEA32ra	r32(W), m(Addr)	-	-	-	MOV32ri
LEA64ra	r64(W), m(Addr)	-	-	-	MOV64ri, XOR64rr
MOV32rr	r32(W), r32(R)	-	-	-	*Delete*, MOV32ri
MOV32rm	r32(W), m32(R)	-	-	-	MOV32ri
MOV32ri	r32(W), i32	-	-	-	-
MOV32mr	m32(W), r32(R)	-	-	-	MOV32mi
MOV32mi	m32(W), i32	-	-	-	-
MOV64rr	r64(W), r64(R)	-	-	-	*Delete*, MOV64ri
MOV64rm	r64(W), m64(R)	-	-	-	MOV64ri
MOV64ri	r64(W), i64	-	-	-	-
MOV64mr	m64(W), r64(R)	-	-	-	MOV64mi
MOV64mi	m64(W), s32	-	-	-	-
MOVAPDrr	x128(W), x128(R)	-	-	-	*Delete*, PXOR128rr
MOVAPDrm	x128(W), m128(R)	-	-	-	PXOR128rr
MOVAPDmr	m128(W), x128(R)	-	-	-	-
MOVSDrr	x64(W), x64(R)	-	-	-	*Delete*
MOVSDrm	x64(W), m64(R)	-	-	-	-
MOVSDmr	m64(W), x64(R)	-	-	-	MOV64mi
MOVUPDrr	x128(W), x128(R)	-	-	-	*Delete*, PXOR128rr
MOVUPDrm	x128(W), m128(R)	-	-	-	PXOR128rr
MOVUPDmr	m128(W), x128(R)	-	-	-	-
MOVUPSrr	x128(W), x128(R)	-	-	-	*Delete*
MOVUPSrm	x128(W), m128(R)	-	-	-	-
MOVUPSmr	m128(W), x128(R)	-	-	-	-
MULPDrr	x128(R/W), x128(R)	-	-	-	MOVAPDrm, MULPDrm, PXOR128rr
MULPDrm	x128(R/W), m128(R)	-	-	-	MOVAPDrm, PXOR128rr
MULSDrr	x64(R/W), x64(R)	-	-	-	MOVSDrm, MULSDrm
MULSDrm	x64(R/W), m64(R)	-	-	-	MOVSDrm
POP16r	r16(W)	RSP(R/W), [RSP](R)	-	-	LEA64ra
POP16m	m16(W)	RSP(R/W), [RSP](R)	-	-	-
POP64r	r64(W)	RSP(R/W), [RSP](R)	-	-	LEA64ra
POP64m	m64(W)	RSP(R/W), [RSP](R)	-	-	-
PUSH16r	r16(R)	RSP(R/W), [RSP - 2](W)	-	-	PUSH16i
PUSH16m	m16(R)	RSP(R/W), [RSP - 2](W)	-	-	PUSH16i

Continued on next page.

Table A.3 – *Continued from previous page.*

PUSH16i	i16	RSP(R/W), [RSP - 2](W)	-	-	-
PUSH64r	r64(R)	RSP(R/W), [RSP - 8](W)	-	-	PUSH64i
PUSH64m	m64(R)	RSP(R/W), [RSP - 8](W)	-	-	PUSH64i
PUSH64i	s32	RSP(R/W), [RSP - 8](W)	-	-	-
PXOR128rr	x128(R?/W), x128(R?)	-	-	-	-
PXOR128rm	x128(R/W), m128(R)	-	-	-	-
RET	-	RSP(R/W), [RSP](R)	-	Return	-
SHL64r	r64(R/W)	CL(R), RFLAGS(W?)	-	-	*Delete*, MOV64ri, SHL64ri, XOR64rr
SHL64ri	r64(R/W), i8	RFLAGS(W?)	-	-	*Delete*, MOV64ri, XOR64rr
SHL64m	m64(R/W)	CL(R), RFLAGS(W?)	-	-	*Delete*, MOV64mi, SHL64mi
SHL64mi	m64(R/W), i8	RFLAGS(W?)	-	-	*Delete*, MOV64mi
SHR64r	r64(R/W)	CL(R), RFLAGS(W?)	-	-	*Delete*, MOV64ri, SHR64ri, XOR64rr
SHR64ri	r64(R/W), i8	RFLAGS(W?)	-	-	*Delete*, MOV64ri, XOR64rr
SHR64m	m64(R/W)	CL(R), RFLAGS(W?)	-	-	*Delete*, MOV64mi, SHR64mi
SHR64mi	m64(R/W), i8	RFLAGS(W?)	-	-	*Delete*, MOV64mi
SUB8rr	r8(R/W), r8(R)	RFLAGS(W)	-	-	-
SUB8rm	r8(R/W), m8(R)	RFLAGS(W)	-	-	-
SUB8ri	r8(R/W), i8	RFLAGS(W)	-	-	-
SUB8mr	m8(R/W), r8(R)	RFLAGS(W)	-	-	-
SUB8mi	m8(R/W), i8	RFLAGS(W)	-	-	-
SUB16rr	r16(R/W), r16(R)	RFLAGS(W)	-	-	-
SUB16rm	r16(R/W), m16(R)	RFLAGS(W)	-	-	-
SUB16ri	r16(R/W), i16	RFLAGS(W)	-	-	-
SUB16mr	m16(R/W), r16(R)	RFLAGS(W)	-	-	-
SUB16mi	m16(R/W), i16	RFLAGS(W)	-	-	-
SUB32rr	r32(R/W), r32(R)	RFLAGS(W)	-	-	-
SUB32rm	r32(R/W), m32(R)	RFLAGS(W)	-	-	-
SUB32ri	r32(R/W), i32	RFLAGS(W)	-	-	-
SUB32mr	m32(R/W), r32(R)	RFLAGS(W)	-	-	-
SUB32mi	m32(R/W), i32	RFLAGS(W)	-	-	-
SUB64rr	r64(R/W), r64(R)	RFLAGS(W)	-	-	-
SUB64rm	r64(R/W), m64(R)	RFLAGS(W)	-	-	-
SUB64ri	r64(R/W), s32	RFLAGS(W)	-	-	-
SUB64mr	m64(R/W), r64(R)	RFLAGS(W)	-	-	-
SUB64mi	m64(R/W), s32	RFLAGS(W)	-	-	-
TEST8rr	r8(R), r8(R)	RFLAGS(W)	-	-	TEST8ri
TEST8ri	r8(R), i8	RFLAGS(W)	-	-	-
TEST8mr	m8(R), r8(R)	RFLAGS(W)	-	-	TEST8mi, TEST8ri
TEST8mi	m8(R), i8	RFLAGS(W)	-	-	-
TEST16rr	r16(R), r16(R)	RFLAGS(W)	-	-	TEST16ri
TEST16ri	r16(R), i16	RFLAGS(W)	-	-	-
TEST16mr	m16(R), r16(R)	RFLAGS(W)	-	-	TEST16mi, TEST16ri
TEST16mi	m16(R), i16	RFLAGS(W)	-	-	-

Continued on next page.

Table A.3 – *Continued from previous page.*

TEST32rr	r32(R), r32(R)	RFLAGS(W)	-	-	TEST32ri
TEST32ri	r32(R), i32	RFLAGS(W)	-	-	-
TEST32mr	m32(R), r32(R)	RFLAGS(W)	-	-	TEST32mi, TEST32ri
TEST32mi	m32(R), i32	RFLAGS(W)	-	-	-
TEST64rr	r64(R), r64(R)	RFLAGS(W)	-	-	TEST64ri
TEST64ri	r64(R), s32	RFLAGS(W)	-	-	-
TEST64mr	m64(R), r64(R)	RFLAGS(W)	-	-	TEST64mi, TEST64ri
TEST64mi	m64(R), s32	RFLAGS(W)	-	-	-
XOR32rr	r32(R?/W), r32(R?)	RFLAGS(W)	-	-	*Delete*, XOR32ri
XOR32rm	r32(R/W), m32(R)	RFLAGS(W)	-	-	*Delete*, XOR32ri
XOR32ri	r32(R/W), i32	RFLAGS(W)	-	-	*Delete*
XOR32mr	m32(R/W), r32(R)	RFLAGS(W)	-	-	*Delete*, XOR32mi
XOR32mi	m32(R/W), i32	RFLAGS(W)	-	-	*Delete*
XOR64rr	r64(R?/W), r64(R?)	RFLAGS(W)	-	-	*Delete*, XOR64rm, XOR64ri
XOR64rm	r64(R/W), m64(R)	RFLAGS(W)	-	-	*Delete*, XOR64ri
XOR64ri	r64(R/W), s32	RFLAGS(W)	-	-	*Delete*
XOR64mr	m64(R/W), r64(R)	RFLAGS(W)	-	-	*Delete*, XOR64mi
XOR64mi	m64(R/W), s32	RFLAGS(W)	-	-	*Delete*

B Compiling and Running the Benchmark

Recent x86-64 Linux distributions can be used to run the benchmark. Packages for tools and libraries such as git, meson, llvm, llvm-devel, and the basic build infrastructure (e.g., make, gcc, g++ ...) are available via the package manager of major Linux distribution. Drob requires a more recent g++ version (e.g., v8.3.0) to compile successfully.

1. Download, compile, and install Drob. The default installation directory is */usr/local/*, which can be overridden via the *PREFIX* environment variable.

```
$ git clone "https://github.com/davidhildenbrand/drob.git"
$ cd drob
$ git checkout 9db20c040d84
$ make
$ sudo make install
```

2. Download DBrew, which includes DBrew-LLVM and a modified benchmark that is also able to benchmark Drob.

```
$ git clone "https://github.com/davidhildenbrand/dbrew.git"
$ cd dbrew
$ git checkout ea3cbb9ad8f9
```

3. Configure the LLVM version by creating a native Meson configuration file *native_config*. The path to the *llvm-config* binary determines the version.

```
[binaries]
llvm-config = "/usr/bin/llvm-config-4.0-64"
```

4. Compile DBrew, DBrew-LLVM, and the benchmark. The *PKG_CONFIG_PATH* environment variable has to be set for the build environment to locate the Drob installation.

```
$ export PKG_CONFIG_PATH=/usr/local/lib64/pkgconfig/
$ meson builddir --native-file=native_config
$ cd builddir
$ ninja
```

5. Run the benchmark using a helper script. Written *.csv* and *.txt* files contain the results.

```
$ python3 benchmark.py --stencil
```

Acronyms

ABI Application Binary Interface.

API Application Programming Interface.

CFG Control Flow Graph.

CFI Control Flow Integrity.

CPU Central Processing Unit.

DBA Dynamic Binary Analysis.

DBI Dynamic Binary Instrumentation.

DBO Dynamic Binary Optimization.

DBR Dynamic Binary Rewriting.

DBT Dynamic Binary Translation.

DFS Depth First Search.

DSP Digital Signal Processor.

FPU Floating Point Unit.

HPC High Performance Computing.

ICFG Interprocedural Control Flow Graph.

IR Intermediate Representation.

ISA Instruction Set Architecture.

JIT Just In Time.

JVM Java Virtual Machine.

PC Program Counter.

RISC Reduced Instruction Set Computer.

ROP Return Oriented Programming.

SBA Static Binary Analysis.

SBI Static Binary Instrumentation.

SBO Static Binary Optimization.

SBR Static Binary Rewriting.

SSA Single Static Assignment.

VLIW Very Long Instruction Word.

VM Virtual Machine.

List of Figures

List of Tables

Listings

Bibliography

[1] LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`. [Online; accessed 29-January-2019].

[2] Machine IR (MIR) Format Reference Manual. `https://llvm.org/docs/MIRLangRef.html`. [Online; accessed 11-May-2019].

[3] The LLVM Target-Independent Code Generator. `https://llvm.org/docs/CodeGenerator.html`. [Online; accessed 23-January-2019].

[4] Valgrind Source Code 3.14.0: VEX documentation. `https://sourceware.org/git/?p=valgrind.git;a=blob;f=VEX/pub/libvex_ir.h;h=17bcb55840f0e33027afb93e9a2f63a98ceb9835;hb=3a3000290b4af0e8ef9880293c54659a6819ba78`. [Online; accessed 29-January-2019].

[5] Wayback Machine: Squeeze++. `https://web.archive.org/web/20041217022620/http://www.elis.ugent.be:80/~brdsutte/squeeze++/download.html`. [Online; accessed 23-January-2019].

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[8] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 295–308. ACM, 2013.

[9] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to Compiler High IR in a binary rewriter. Technical report, University of Maryland, Maryland, USA, 2010.

[10] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, volume 31, pages 149–159. ACM, 1996.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent Dynamic Optimization: The Design and Implementation of Dynamo (HPL-1999-78). Technical report, HP Laboratories, Cambridge, UK, 1999.

[12] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2000.

[13] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[14] Fabrice Bellard. QEMU Source Code v3.1.0: Tiny Code Generator Documentation. `https://git.qemu.org/?p=qemu.git;a=blob;f=tcg/README;h=d22ee084b83698a1c68ce68e9167cdc5846b12ea;hb=32a1a94dd324d33578dca1dc96d7896a0244d768`. [Online; accessed 29-January-2019].

[15] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*. USENIX Association, 2005.

[16] Matthias Braun. Welcome to the Back End: The LLVM Machine Representation. `http://llvm.org/devmtg/2017-10/slides/Braun-Welcome%20to%20the%20Back%20End.pdf`, 2017. [Online; accessed 11-May-2019].

[17] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275. IEEE, 2003.

[18] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, pages 133–144. ACM, 2012.

[19] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[20] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469. Springer-Verlag, 2011.

[21] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[22] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 137–147. ACM, 2007.

[23] Filipe Cabecinhas, Nuno Lopes, Renato Crisostomo, and Luis Veiga. Optimizing Binary Code Produced by Valgrind (RT/46/2008). Technical report, INESC-ID/IST, Distributed Systems Group, Lisboa, Portugal, 2008.

[24] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo, and Rémy Gauguey. deGoal a tool to embed dynamic code generators into applications. In *Proceeedings of the 23rd International Conference on Compiler Construction*, pages 107–112. Springer, 2014.

[25] W. K Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A Dynamic Optimization System. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

[26] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 2010.

[27] Robert Cohn, David Goodwin, P Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*, pages 17–24. USENIX Association, 1997.

[28] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1078–1083. EDA Consortium / IEEE, 2015.

[29] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-Time Binary Rewriting Techniques for Program Compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 2005.

[30] Dean Deaver, Rick Gorton, and Norm Rubin. Wiggins/Redstone: An On-line Program Specializer. In *Hot Chips 11 - A Symposium on High performance chips*, 1999.

[31] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):378–415, 2000.

[32] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141. ACM, 2017.

[33] Jiun Hung Ding, Po Chun Chang, Wei Chung Hsu, and Yeh Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *Proceedings of the 17th International Conference on Parallel and Distributed Systems*, pages 276–283. IEEE, 2011.

[34] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[35] Alexis Engelke and Josef Weidendorfer. Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime. In *Proceedings of the 31st International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 785–794. IEEE, 2017.

[36] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 131–144. ACM, 1996.

[37] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 75–88. USENIX Association, 2006.

[38] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proceedings of the NewSecurity Paradigms Workshop*, pages 87–95, 1999.

[39] Alessandro Di Federico. *Compiler Techniques for Binary Analysis and Hardening.* PhD thesis, Politecnico di Milano, 2018.

[40] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Politecnico Milano. How the ELF Ruined Christmas. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 643–658. USENIX Association, 2015.

[41] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1-2):147–199, 2000.

[42] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An Evaluation of Staged Run-Time Optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 293–304. ACM, 1999.

[43] Jim Grosbach and Owen Anderson. LLVM MC In Practice. LLVM Developers' Meeting: `https://llvm.org/devmtg/2011-11/Grosbach_Anderson_LLVMMC.pdf`, 2011. [Online; accessed 29-January-2019].

[44] Nabil Hallou. *Runtime Optimization of Binary Through Vectorization Transformations.* PhD thesis, Université de Rennes 1, 2017.

[45] Nabil Hallou, Erven Rohou, and Philippe Clauss. Runtime Vectorization Transformations of Binary Code. *International Journal of Parallel Programming*, 45(6):1536–1565, 2017.

[46] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. Dynamic Re-Vectorization of Binary Code. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 228–237. IEEE, 2015.

[47] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. Zipr: Efficient Static Binary Rewriting for Security. In *Proceedings of*

the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 559–566. IEEE, 2017.

[48] Christian Heitman and Iván Arce. BARF: A multiplatform open source Binary Analysis and Reverse engineering Framework. In *XX Congreso Argentino de Ciencias de la Computación*, 2014.

[49] Ealan A Henis, Gadi Haber, Moshe Klausner, and Alex Warshavsky. Feedback Based Post-link Optimization for Large Subsystems. In *Proceedings of the 2nd Workshop on Feedback Directed Optimization*, pages 13–20, 1999.

[50] J. Hiser, A. Nguyen-Tuong, W. Hawkins, M. McGill, M. Co, and J. Davidson. Zipr++: Exceptional Binary Rewriting. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 9–15. ACM, 2017.

[51] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 104–113. ACM, 2012.

[52] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 2011.

[53] Jan Hubička, Andreas Jaeger, Michael Matz, and Mark Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement. `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`, July 2013. [Online; accessed 18-April-2019].

[54] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*. USENIX Association, 1999.

[55] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. DisIRer: Converting a Retargetable Compiler into a Multiplatform Binary Translator. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4), 2010.

[56] Wen Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1-2):229–248, 1993.

[57] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual`, April 2018. [Online; accessed 27-March-2019].

[58] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4`, January 2019. [Online; accessed 27-March-2019].

[59] Daniel Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 63–80. Springer-Verlag, 2001.

[60] Daniel Kästner and Stephan Wilhelm. Generic Control Flow Reconstruction from Assembly Code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, pages 46–55. ACM, 2002.

[61] Jinpyo Kim, Wei Chung Hsu, and Pen Chung Yew. COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 2007.

[62] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, Dongyeop Oh, Jonghyup Lee, and Sang Kil Cha. Testing Intermediate Representations for Binary Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 353–364. IEEE, 2017.

[63] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 412–424. ACM, 2017.

[64] Kevin Kirchner and Stefan Rosenthaler. bin2llvm: Analysis of Binary Programs Using LLVM Intermediate Representation. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017.

[65] Thomas Kistler and Michael Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, 2001.

[66] Thomas Kistler and Michael Franz. Continuous Program Optimization: A Case Study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.

[67] Toshihiko Koju, Reid Copeland, Motohiro Kawahito, and Moriyoshi Ohara. Reconstructing High-Level Information for Language-Specific Binary Re-optimization. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 273–283. ACM, 2016.

[68] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic Parallelization in a Binary Rewriter. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 547–557. IEEE, 2010.

[69] James R Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 291–300. ACM, 1995.

[70] Chris Lattner. Intro to the LLVM MC Project. `http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html`, 2010. [Online; accessed 23-January-2019].

[71] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[72] Chris Arthur Lattner. *LLVM : An Infrastructure for Multi-Stage Optimization*. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[73] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Processdings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 175–183. IEEE, 2010.

[74] Anthony Liguori. ANNOUNCE: Release 0.10.0 of QEMU. `http://lists.gnu.org/archive/html/qemu-devel/2009-03/msg00154.html`, 2009. [Online; accessed 23-January-2019].

[75] Cullen Linn, Saumya Debray, Gregory Andrews, and Benjamin Schwarz. Stack Analysis of x86 Executables. *Manuscript*, 2004.

[76] Sheldon Lobo. The Sun Studio Binary Code Optimizer. `https://www.oracle.com/technetwork/server-storage/solaris10/binopt-136601.html`, 2015. [Online; accessed 25-January-2019].

[77] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. *Journal of Instruction-Level Parallelism*, 6:1–24, 2004.

[78] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200. ACM, 2005.

[79] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–26. IEEE, 2004.

[80] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. DBILL: An Efficient and Retargetable Dynamic Binary Instrumentation Framework using LLVM Backend. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 141–152. ACM, 2014.

[81] Daryl Maier. Testarossa's Intermediate Language: An Intro to Trees. `https://github.com/eclipse/omr/blob/master/doc/compiler/il/IntroToTrees.md`, 2016. [Online; accessed 23-January-2019].

[82] Amir Majlesi-Kupaei, Danny Kim, Kapil Anand, Khaled ElWazeer, and Rajeev Barua. RL-Bin, Robust Low-overhead Binary Rewriter. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 17–22. ACM, 2017.

[83] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Bronis de Supinski, Sally McKee, and Andy Yoo. METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 289–300. IEEE, 2003.

[84] Florian Märkl. *Case Study on LLVM as suitable intermediate language for binary analysis*. Bachelor seminar thesis, Technische Universität München, 2017.

[85] Cathy May. Mimic: A fast System/370 simulator. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*. ACM, 1987.

[86] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium*. USENIX Association, 2006.

[87] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. alto: A Link-Time Optimizer for the DEC Alpha. Technical report, University of Arizona, Tucson, AZ, USA, 1998.

[88] Takashi Nakamura, Satoshi Miki, and Shuichi Oikawa. Automatic Vectorization by Runtime Binary Translation. In *Proceedings of the 2nd International Conference on Networking and Computing*, pages 87–94. IEEE, 2011.

[89] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 358–369. IEEE, 2006.

[90] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[91] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[92] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis. Retrofitting Security in COTS Software with Binary Rewriting. In *Proceedings of the International Information Security Conference*, pages 154–172. Springer, 2011.

[93] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14. IEEE, 2019.

[94] Mathias Payer, Boris Bluntschli, and Thomas R. Gross. DynSec: On-the-fly Code Rewriting and Repair. In *Proceedings of the 5th International Workshop on Hot Topics in Software Upgrades*, 2013.

[95] Manish Prasad and Tzi-cker Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 211–224. USENIX Association, 2003.

[96] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *Proceedings of the International Workshop on Dynamic Compilation Everywhere*, 2014.

[97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*. USENIX Association, 1997.

[98] Yukinori Sato, Tomoya Yuki, and Toshio Endo. ExanaDBT: A Dynamic Compilation System for Transparent Polyhedral Optimizations at Runtime. In *Proceedings of the Computing Frontiers Conference*, pages 191–200, 2017.

[99] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *Proceedings of the Workshop on Binary Translation*, 2001.

[100] Benjamin William Schwarz. *Post Link-Time Optimization on the Intel IA-32 Architecture*. PhD thesis, University of Arizona, 2002.

[101] Julian Seward. The design and implementation of Valgrind. `https://courses.cs.washington.edu/courses/cse326/05au/valgrind-doc/mc_techdocs.html`, 2002. [Online; accessed 23-January-2019].

[102] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

[103] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2005.

[104] Noah Snavely, Saumya Debray, and Gregory Andrews. Predicate Analysis and If-Conversion in an Itanium Link-Time Optimizer. In *Proceedings of the Workshop on Explicitly Parallel Instruction Set (EPIC) Architectures and Compilation Techniques (EPIC-2)*, 2002.

[105] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, pages 1–25. Springer-Verlag, 2008.

[106] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: A Low-Overhead Dynamic Translator. *ACM SIGARCH Computer Architecture News*, 35(1):135–140, 2007.

[107] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: An Open Source, Low-Level Dynamic Instrumentation System. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 175–185. ACM, 2006.

[108] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical report, Microsoft Corporation, Redmond, WA, USA, 2001.

[109] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Program Analysis Customized Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[110] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. Technical report, Western Research Laboratory, Palo Alto, California, USA, 1992.

[111] James Stanier and Des Watson. Intermediate Representations in Imperative Compilers: A Survey. *ACM Computing Surveys*, 45(3), 2013.

[112] Mark Stoodley. Under the Hood of the Testarossa JIT Compiler. `https://www.slideshare.net/MarkStoodley/under-the-hood-of-the-testarossa-jit-compiler`, 2016. [Online; accessed 23-January-2019].

[113] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic Native Optimization of Interpreters. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 50–57. ACM, 2003.

[114] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, pages 7–12. IEEE, 2005.

[115] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security*, 2017.

[116] Shuai Wang, Pei Wang, and Dinghao Wu. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pages 236–247. IEEE, 2016.

[117] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[118] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 299–308. ACM, 2012.

[119] Josef Weidendorfer and Jens Breitbart. The Case for Binary Rewriting at Runtime for Efficient Implementation of High-Level Programming Models in HPC. In *Proceedings of the 30th International Parallel and Distributed Processing Symposium*, pages 376–385. IEEE, 2016.

[120] Curtis Yarvin and Adam Sah. QuaC: Binary Optimization for Fast Runtime Code Generation in C. Technical report, EECS Department, University of California, Berkeley, USA, 1994.

[121] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[122] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 129–140. ACM, 2014.

[123] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX conference on Security*, pages 337–352. USENIX Association, 2013.