Towards Verified Binary Raising

Joe Hendrix[1], Guannan Wei[2], and Simon Winwood[1]

[1] Galois, Inc `{jhendrix,sjw}@galois.com`
[2] Purdue University `wei220@purdue.edu`

Abstract

We provide a high-level overview of two tools developed at Galois on generating verified LLVM from binaries: `reopt`, a binary recompilation framework lifts portions of a binary into LLVM, optimizes the LLVM, and then generates a new binary; and `reopt-vcg` is a standalone tool capable of verifying that the behaviors in the LLVM are allowed in the source binary. The goals of this effort are to build tools for transforming existing binary software post-relase to do things like apply mission or environment specific optimizations or introduce additional security hardening techniques. As these tools will rewrite binaries after traditional testing is complete, a high-level of assurance is required.

1 Introduction

Compiler verification has long been a topic of interest within the formal methods community. Typically one wants to show that every execution of the compiled program represents behavior allowed in the source program. The converse need not be true: there may be source program behaviors that the binary program never exhibits. For example, a source language such as C may permit the function arguments to be evaluated in any order while the compiler will pick a specific order during compilation.

Program decompilation, that is analyzing the compiled binary to infer a possible source program that could have generated it, is a widely used technique for reverse engineering. Recently this technique has been used to assist in verifying compiled artifacts and to recompile applications for increased performance or security protections. In these latter applications, one does not need a full decompiler back to program source, and hence *binary raising* into an intermediate representation such as assembly code or LLVM is sufficient.

When raising a binary for rewriting purposes, it is imperative that the decompilation process does not itself introduce new behaviors in the binary, as the program rewriting will typically be done post-development, and will not have access to the testing and verification suite used to originally develop the applications. In contrast to compiler verification (e.g., [3]), the binary recovery process is not given the program structure but rather must infer it using techniques such as value-set analysis.

In this paper, we describe ongoing work on binary raising at Galois. In Section 2, we describe the `reopt` program reoptimization tool which raises binaries into to LLVM for recompilation. In Section 3 we describe our translation verifier `reopt-vcg` for checking the correctness of the generated LLVM.

Related Work. Decompilation and the underlying techniques have been extensively studied, and due to space limitations we only survey binary raising tools that target LLVM. These include SecondWrite [1], McSema [6], RevGen [2], and `MCTOOL`[8]. Reopt's design is close to SecondWrite, but it is not publicly available, and so we are unable to comprehensively evaluate it. McSema and RevGen use a more direct translation that maps processor registers to a

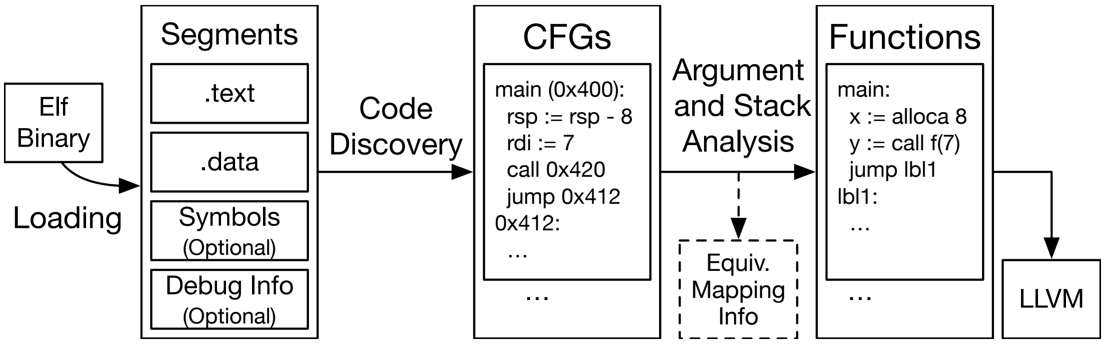

Figure 1: Binary recovery

struct; this simplies the implementation, but results in LLVM that does not preserve the binary interfaces and instead uses an explicit and implicit stack. MCTOOL is based heavily on LLVM, and appears also to generate ABI-compatible functions.

The above tools do not provide assurance that the LLVM is correct for a given binary. Myreen et al. [4] translate blocks of ARM machine code into HOL4 terms that capture the semantics of the block. This work simplifies the verification of machine code within a theorem prover; Sewell et al. [7], use this framework to prove translation validation for C.

2 Binary Raising

Reopt[1] is a post-compilation optimization tool that lifts functions in a compiled binary into corresponding LLVM bitcode, runs the LLVM optimizer, and then recombines the LLVM with regions of the original binary, including data and code that could not be lifted, to produce a new executable. This allows one to use LLVM optimization passes across object code boundaries, or add additional compiler-provided protections to code that was compiled without it. Reopt currently supports static X86_64 binaries compiled in the ELF format for Linux. For this paper, we focus exclusively on the binary to LLVM raising process as that is most closely related to the formal specification of instruction set architectures.

Reopt's binary raising occurs as a three step process as shown in Figure 1. The dotted lines to equivalence mapping info indicates that feature in development to support `reopt-vcg`. The basic process consists of (1) loading the binary, (2) a code control-flow discovery algorithm (discussed below) to identify candidate function entry points as well as basic blocks within each function, (3) a global analysis to compute register and stack usage to infer function prototypes and high-water marks on the stack.

Reopt has been released under a BSD-derived license, and is available on Github [2]. In Reopt, the LLVM generated from a binary is sent to LLVM's optimizer `opt`, and then combined with the existing code and data to generate a new executable.

Instruction Set Formalization. Reopt uses an open-source library developed by Galois, Macaw[3], for code discovery. Macaw is written to support multiple architectures, and has

[1]Available at https://github.com/Galoisinc/reopt
[2]See https://github.com/GaloisInc/reopt
[3]Available at https://github.com/Galoisinc/macaw

libraries for both X86_64 and PowerPC available. The architecture-specific libraries are responsible for declaring the architecture-specific concepts including the defining the processor register set, defining specific operations that have side effects or raise exceptions such as floating point operations and system-level instructions, and providing information about ABI concepts such as calling conventions and jump-table layout.

The architecture support libraries provide a semantics for the given architecture in the form of operations for converting byte streams into basic blocks. A basic blocks is represented as a sequence of simple register transfer language statements. For X86_64, we have manually written the instruction semantics and cover many common instructions involving general purpose registers, but only a small portion of traditional x87 operations, system-level instructions, and SSE and AVX instructions.

3 Verification

In binary rewriting, it is imperative that any changes to the binary's behavior are intended. The first step of binary raising should not make any changes, but only lift the application. As part of this work, we have constructed a translation verifier, `reopt-vcg` that takes a binary executable, LLVM bitcode file, and annotations that relate LLVM functions to addresses in the executable, and generates proof obligations in the SMT-LIB over the theory of arrays, bitvectors and uninterpreted functions (`QF_AUFBV`).

Our verifier seeks to establish that each observable behavior in the LLVM has a corresponding machine code behavior. For our purposes, we use a strong form of equivalence in which each operation with side effects, including memory reads and writes in the LLVM, has an equivalent write in the machine code. For most operations the equivalence must be exact (e.g., in a divison operation that may signal due to a divide by zero, the divisors must be equivalent). The one exception however is memory accesses to the stack. LLVM does not allow us to control stack layout precisely, and omits some of the memory accesses to the stack present in machine code such as storing the frame pointer and the explicit push of the return address to the stack in a function call. Finally and most severely, we do not currently allow pointers to the stack to be stored in global variables or passed between functions. The latter would require additional memory safety checks for soundness that we do not yet support.

Our verifier accounts for these through annotations and several simplifying assumptions that are satisfied by Reopt. First, each LLVM basic block must correspond to a particular contiguous set of bytes in the machine code program, and moreover operations with side effects such as memory reads and writes that correspond must have the same order within those blocks. Furthermore, we require the annotations distinguish stack from heap accesses and identify the correspondance between accesses.

Each basic block within the functions is verified independently, and the annotations are used to define the appropriate pre-conditions for the block with the function entry point preconditions defined by annotations on the function along with requirements imposed by the platform ABI on how arguments map to machine code registers. We have tested the verifier on several small programs with manually generated annotations, and are currently working on automating the annotation generation process.

4 Future Work

The current implementation of `reopt-vcg` is written in Haskell. We are currently working on implementing this tool in the Lean theorem prover, along with a proof that the VCGs entail the equivalence property described above. The intent then is to have a verified translation verifier for the binary raiser. The key challenge is to prove that our block level verifications are compositional, so that one has an end-to-end proof of the entire execution.

We currently support a subset of the X8_64 instruction set in Lean; we are experimenting with using semantics developed by others to both increase the instruction set coverage, and to avoid institutional bias in the definition of the semantics. We have done experiments in this direction both by leveraging Valgrind's VEX IR [4], and ASL [5]. With Valgrind, we have been able to use the semantics to decode binaries, but there is a large number of operations in Valgrind's micro-op language and their semantics is never documented and often unclear. Our ASL work is still in its early stage and shows promise; a key challenge will be deciding when to expand ASL functions into their definitions, and when to map the functions to higher-level primitives. Some of the ASL definitions contain loops or conditional control flow, and expanding the definition may lead to less efficient code if LLVM optimizations are not able to reconstruct the original more efficient encoding.

Acknowledgements. We thank Steve Zdancewic, Jason Dagit, and Rob Dockins for discussions and software contributions. We also thank our sponsors at the Office of Naval Research. This work was sponsored by the Office of Naval Research under Contract No. N68335-17-C-0558. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research.

References

[1] 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 Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 295–308. ACM, 2013.

[2] Vitaly Chipounov and George Candea. Enabling sophisticated analyses of ×86 binaries with revgen. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011), Hong Kong, China, June 27-30, 2011.*, pages 211–216. IEEE Computer Society, 2011.

[3] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[4] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic - improved. In *FMCAD 2012*, pages 78–81. IEEE, 2012.

[5] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with isa-formal. In *Computer Aided Verification (CAV 2016)*, volume 9780 of *LNCS*, pages 42–58. Springer, 2016.

[6] Andrew Ruef and Artem Dinaburg. Static translation of X86 instruction semantics to LLVM with McSema. Presented at REcon 2014; Slides at https://github.com/trailofbits/mcsema, 2014.

[7] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation, PLDI '13*, pages 471–482. ACM, 2013.

[8] S. Bharadwaj Yadavalli and Aaron Smith. Raising binaries to LLVM IR with MCTOLL. Upcoming presentation PLDI 2019; Software at https://github.com/microsoft/llvm-mctoll, 2019.

[4] Available at http://valgrind.org