Proseminar Software-Hardening, SoSe 2025

VEX vs. P-Code: Intermediate Representations in the Reverse-Engineering Platforms angr and Ghidra

Marcel Fiedler
Matr.Nr.: 777
E-Mail: deine.mudder@campus.lmu.de
Ludwig-Maximilians-Universität München
Institut für Informatik

Abstract

Comparing binaries from different platforms and architectures is a necessary but complicated task in software hardening and malware analysis. Intermediate Representation offers a way of representing programming logic by rewriting it into an abstraction that lies between the source language's structure and the target assembly's concrete structure, while being detached from the limitations of an actual architecture. Reverse engineering platforms and compilers use it to further analyze it, such as optimization, analysis, and efficient code generation.

This approach also enables a better development of novel solutions: It is often easier and faster to create a lifter into an IR for a new architecture than to write an individual virtual machine for it.

This paper compares the IR of the popular reverse engineering platforms VEX from angr and P-Code from Ghidra. The focus lies on the different approaches of the IR, their creation and usage by the reverse engineering platforms, and their performance with varying analysis approaches.

1 Introduction

The term reverse engineering originates from hardware analysis, which aims to decipher the architecture of established products.[4] The reverse engineering process of software is the identification of the components of the system and their collaboration. The next step is to create an abstract representation of the system: The Intermediate Representation (IR).

Reverse engineering platforms and compilers use it to do further analysis with it, such as optimization, analysis, and code generation to higher-level languages such as C.

Intermediate representations, also known as intermediate languages, are designed to remove this burden.

IR offers a way of representing programming logic by rewriting it into an abstraction between the structure of the high-level language source and the concrete structure of the target assembly, while detached from the limitations of an actual architecture.

Concrete assembly instruction sets contain complex semantics with different side effects, such as setting status flags. The overwhelming quantity of those instructions makes automatic binary analysis complicated.

The use of IR offers more flexibility and saves time. It can also be used in a broader range of tools as an interface for many other binary analyses on top of the translated code. Instead of writing a specific handler for every binary analysis for each architecture, it only has to be done once using IR.

Reverse engineering offers a way to understand binary code from other authors and decide whether they are malicious or not, and as an offensive practice, develop methods to protect operating systems and crucial software against those attacks.

But as the influence of technology widens even further and with an increasing usage of computer-driven tools in all aspects of industries and private lives in sight, the possibilities for software exploitation rise, as more and more of everyday life and critical infrastructure becomes influenced by computers.

Malware authors adapt to defense techniques with new attacks, crafting even more complicated ways of exploitation and making the binaries obfuscated to evade detection during the analysis process. These practices are called anti-disassembly: Code and data in programs to cause disassembly analysis to make incorrect program listings to veil the malicious intention of the program. Malware authors also use anti-disassembly techniques to disrupt the analysis of malicious code; some of these techniques are: anti-disassembly, anti-debugging, anti-virtual machine techniques, and binary packing.[13] The variety of anti-reverse engineering techniques illustrates the need for more research and development in reverse engineering.

2 Background

2.1 What is Reverse Engineering?

Software reverse-engineering is the analysis of systems in order to identify their components and their interrelationships. Furthermore, to make a representation of the system in another form or abstraction.[4] The IEEE Standard for Software Maintenance defines reverse engineering as ”the process of extracting software system information from source code.”[1] It can, however, start at any level of abstraction and any stage of the software development life cycle. The goal of the examination is the understanding of the logic behind a program and deciding whether it is harmful or not. The main domains of reverse engineering are *Redocumentation* and *Design recovery*.

Redocumentation
Redocumentation is the recreation of a ”semantically equivalent representation within the same relative abstraction level”[4, p.15], e.g., dataflow, data structures, control flows, for a better oversight. It is the simplest and oldest form of reverse engineering. The goal is to provide an easily understandable visual documentation of the program components.[4]

Design recovery
Design recovery is the gaining of insights of higher-level abstractions beyond the direct observation itself by combining domain- and external information as well as domain knowledge or fuzzy reasoning.[4]

Restructuring
Restructuring means changing one form of representation into another on the same level of abstraction, all while preserving the behavior of the software.[4]

Reengineering
Reengineering is the analysis of a program to alter it into a new form.

2.2 Disassembly vs. Decompilation

Disassembling
Reverse engineering with disassembly is the translation of binaries in machine code into human-understandable assembly. The caveat of the faster execution of compiled languages in comparison to interpreted ones is the loss of meta-information from the original program, e.g., lost names of variables and functions; therefore, complete reversion is not possible. However, a disassembler can trace back the execution and display it as human-readable assembly mnemonics.[8]

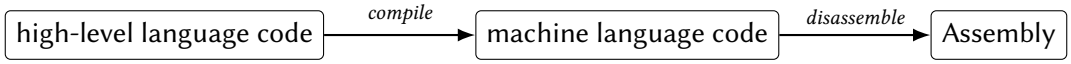

Figure 1: Disassembly workflow[8]

Decompile
Interpretable languages such as Java and Python are not compiled into machine code, but into *byte code*, which gets interpreted at runtime by a virtual machine. Despite some optimization, the context is still preserved, which makes the reverse translation into the original high-level language code possible.[8]

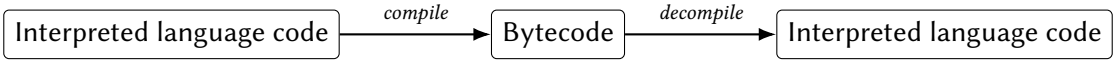

Figure 2: Decompilation workflow[8]

2.3 Platforms for Reverse Engineering and Binary Analysis

Binary analysis is the main reverse engineering practice: The examination at the binary code level to uncover eventual vulnerabilities and threats when the original higher-language source code from which it was compiled is unavailable. In order to perform binary analyses and other reverse engineering techniques, a variety of platforms have been developed that offer an efficient and professional way of analyzing binaries.

2.4 What are Intermediate Representations?

Intermediate Representations are used internally by compilers, decompilers, and virtual machines. IR translates the machine code of concrete architectures into a more abstract representation of its program logic. IR is used for optimization, analysis, and efficient code generation. IR usually has an *external format* which can be used as an interface for other unrelated tool that enables reverse-engineering platforms. There are many types of IR. This term paper will present the *static single assignment form* and *control flow graph.*[15]

Static Single Assignment (SSA)
is a common used form for IR with static variables, commonly used for complex optimizations. The variables in the code blocks of the control flow graph have the restriction that variables canot

be updated. Instead the variable is replaced by a new version:[15] The left side of every variable assignment must have a unique name. The output of each basic code block is determined by an individual ϕ-function which selects an output out on its input operands based on which control flow edge the dynamic execution entered the basic block through.[19]

Control Flow Graph (CFG)
CFG representation is used to show the higher level structure of a program. CFGs are directed graphs whose nodes are *basic blocks*,i.e. maximal sequences of statements with a single entry and single exit. The edges between them represent the decision flow of sequential execution. Conditional constructs appear as separate blocks and result in branch nodes in the graph that split the control flow, while loops are represented by cycles in the graph.[15]

3 Reverse Engineering Internals

3.1 angr internals

The first step into reverse-engineering with angr is loading a binary into a project, which consists of information about its CPU architecture, filename, and the entry point address.[17] The binary is then loaded into the angr analysis system by the *CLE loader* by providing base classes for reproducing binary objects. CLE uses *file format pasing libraries*, e.g. `elftool` for Linux and `portable executable` files for Windows, to parse the objects themselves. Then the *memory image* is exposed by reloactions in the binary file.[12] The `SimuVEX` module creates a *program state*, a snapshot of registers, memory, and open files, which is then used by angr's simulation engine `SimEngine`. `SimuVEX` uses a set of plugins such as `project.factory.block()` to extract *basic blocks* from the binary into *block objects*, which saves the entry point of the code block, assembly mnemonics, an instruction counter, as well as addresses of the instructions.[12, 17] This data is then used as abstract expressions in a *expression tree* with values as leaf nodes and operations as non-leaf nodes.
The functionality of Claripy is divided into two parts, the *frontend* and *backend*: The expressions can be translated by the backend into data domains at any moment in the examination process. It supports the concrete domain (integers, floating-point numbers) and the symbolic domain (symbolic integers and floating-point numbers). The frontend modules provide augmentation for the backend with additional functionality, such as *symbolic solving, constraint solving, expression interpretation, symbolic constraint solving.* Angr provides several analyses, e.g., *dynamic symbolic execution, CFG recovery.* The main interfaces for such analyses are `path groups`, which are modules that are interfaces to *dynamic symbolic execution.* They track *execution paths* of applications. `analyses` provides abstractions for full program analyses and manages static analyses.

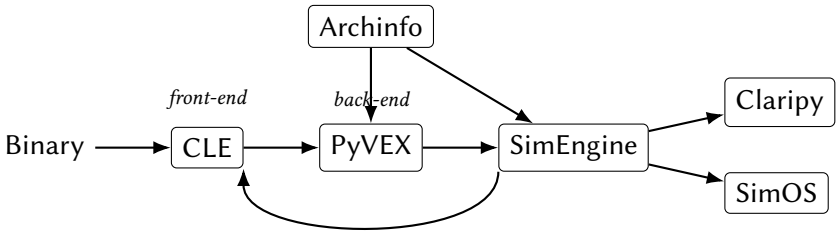

Figure 3: Internal workflow of angr.[14]

CLE loader
The CLE loader (CLE stands for "CLE loads everything") extracts the actual executable from the binary code. It can read many types, such as portable executable, executable and linking format binaries. The CLE loader is a set of *binary objects* which are loaded and mapped into a single memory space. Each filetype is processed by a specialized loader backend.[18] Depending on the header of the file, CLE then makes assumptions on the aimed architecture of the binary. It creates a representation of the *memory map*, the segment of the virtual memory which has been assigned a direct byte-for-byte correlation with some part of the file, by simulating the actual loader. The result is a Loader object with the memory of the program.

Archinfo
During the loading process, the targeted architecture of the binary is guessed by analyzing the header or other heuristics. With this information, the `archinfo` library is called. This contains a map of the register file, bid width, usual endian-ness, etc.

SimEngine
The `SimEngine` interprets the code in a meaningful way. It saves the *programs state*, a snapshot of the registers, memory, basic block of instructions, etc., and produces a set of *successors:* Possible data program states that can be reached from this block. When branches occur, the *constraints*, conditions which are needed to take paths of branches, are collected.

PyVEX
The default engine of the lifer, `SimEngineVEX`, supports many architectures because it does not run directly on their machine code. Instead, it uses the `VEX IR`, which the machine code is lifted into.[14]
Instead of creating a new engine for a special architecture, a lifter can be a more efficient solution: It allows PyVEX to take care of the rest.[12]

Claripy
Angrs lifter Claripy is an abstracted constraint-solving wrapper[16]. Operations are not represented as concrete values, but as expressions which are eventually composed from several expressions them self. Claripy creates these compositions and solves them with Microsofts SMT-solver Z3.[14]

SimOS
Higher level functionality and abstraction such as `stdin`, file management, networking are done by the OS. SimOS provides an abstraction for this by defining OS-specific embellishments on the initial state of the program. system calls and symbolic summaries of syscalls and common

library functions are provided via `SimProcedures`. This makes angr faster and compatible than symbolically executing `libc` itself.

3.2 Ghidra internals

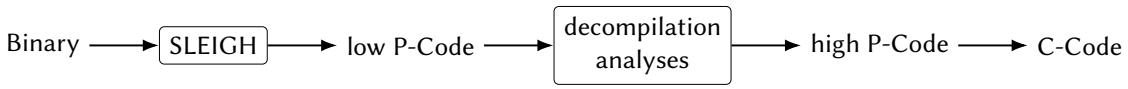

Figure 4: Internal Ghidra workflow.[9]

The first step of the Ghidra reverse engineering process is the binary disassembly with *SLEIGH*.[9]
SLEIGH is a *processor specification language* for Ghidra. It is derived from *SLED (specification language for encoding and decoding)*. It is a language for describing an instruction set of general purpose microprocessors and detailed enough to facilitate the *disassembly-* and *decompilation engines*.
For disassembly, SLEIGH offers a concise description of the translation from bit encoding of the machine instructions to readable assembly code. The single operands get divided into mnemonic operands, sub-operands and associated syntax.[10]
As for the decompilation, SLEIGH describes the rules for the translation from machine instructions into p-code. It describes the specification of the encoding for processors with `.slaspec`-files.[10, 7]

4 Comparison of the IR types

4.1 VEX

The VEX IR (Virtual Execution eXtended Intermediate Representation) was developed as part of *Valgrind*, which was developed in 2000 by Julian Seward as a memory debugging tool but has since evolved into a modular framework for binary analysis for various architectures. Its core design is dynamic binary translation which allows the analysis of every instruction executed in a program. In contrast to angr which uses the VEX IR for *static-* and *dynamic symbolic execution, CFG-recovery and vulnerability detection.* Valgrind focuses on runtime correctness checking and profiling, angr is designed for program reasoning, reverse engineering and automated exploit discovery as a broader and more research-oriented development process. VEX IR represents register content as separate memory spaces with integer offsets, enables memory segmentation and is free of side-effects. It represents the functionality of processors as four objects: *expressions, operations, temporary variables, statements* and *blocks*

Expressions
Content of memory and registers and their results of *IR operations*.

Operations
IR Operation represent the modifications on the IR expression-values. The data types include integers, arithmetic, floating-point numbers, bit-operations.

Temporary variables
Data may be stored as strongly typed temporary variables in *internal registers*, which can be read

with IR operations.

Statements
IR Statements are keeping track of changes in the state of targeted processor architecture such as read and write in memory and registers. The data which they use are retrieved by the IR expressions.

Blocks
IR blocks represent *IR super blocks (IRSB)* of the targeted architecture and are a set of IR statements.

4.2 P-Code

Ghidra uses *P-Code* as IR. It is a *register transfer language* general enough to abstract the machine code of several general purpose processors. It enables explicit data manipulation on user defined register and address spaces. During the analysis process, the p-code also gets lifted into *single static assignment (SSA)*[3]. During the SSA optimization, all code blocks get updated from gained data from the analysis of the control flow. However whenever the value of a variable changes it does not get updated, instead it is replaced by a new version number.[15] P-Code mirrors processor tasks and concepts with three core ideas: *Address space, Varnodes* and *Operations.*

Address Spaces
The P-Code address space is a generalization of RAM. It is defined as a sequence of bytes which can be read and written by P-Code operations and are characterized by name and size. Only in the address spaces data can be manipulated.

Varnodes
Registers of a processor are represented by varnodes. They are saved as a continuous set of bytes in a p-code address space and get manipulated by P-Code operations. Varnodes are typeless sequential bytes, only characterized by a starting address and size. The p-code operations can then interpret them as types, which are integer, boolean and floating point numbers.

Operations
P-code operations represent the machine instructions of a processor. All operations are mapped to the address of a specific machine instruction and can be generalized as functions which take one or more varnodes as input and exactly one varnode as output:[2]

$$\text{P-Code operation}(v_0, v_1, \ldots, v_{n\in\mathbb{N}}) \rightarrow v_{new}$$

5 Performance comparison

VEX IR was developed by the valgrind project with an architecture agnostic approach, focusing on abstraction and quick program analysis, which may be not human readable but optimized for automatization and symbolic execution.
P-Code focuses mainly on static analysis and decompilation, and has a more human readable semantics and a clearer mapping from machine code to intermediate representation.
The following comparison is regarding the difference in performance measuring outcomes of the papers: *Decompilebench: A comprehnsible benchmark for evaluating decompilers in the real world*

scenarios[5], *DisCo: Combining Disassemblers for Improved Performance*[11] and *An Empirical Study on ARM Disassembly Tools*[6]

5.1 Decompilation

`DecompileBench` is the first comprehensible framework for effective evaluation of compilers in reverse engineering workflows, focusing on semantic accuracy and analyst usability of decompilation.
While the paper focuses on the possibility of the usage of LLM-powered approaches for reverse engineering, the only takeaway for this seminar-paper will be the IR benchmark comparisons.

The table below shows the *recompile success rate (RSR)* and *coverage equivalence rate (CER)* of angr and Ghidra with several compiler optimization stages.
The RSR is the percentage of decompiled functions that can be successfully recompiled back into valid machine code it measures if the output is able to compile: If the decompiler produces code that the compiler accepts and turns into an executable without errors, it counts as a success.
CER measures how many recompiled functions preserve the original behavior at runtime and produce a equivalent execution outcome.

	Recompile Success Rate						**Coverage Equivalence Rate**					
Decompiler	**O0**	**O1**	**O2**	**O3**	**Os**	**Avg**	**O0**	**O1**	**O2**	**O3**	**Os**	**Avg**
Angr	0.309	0.232	0.190	0.181	0.191	**0.221**	0.187	0.153	0.124	0.116	0.118	**0.140**
Ghidra	0.524	0.421	0.395	0.377	0.353	**0.413**	0.374	0.294	0.256	0.241	0.228	**0.278**

Table 1: Comparison of Angr and Ghidra for Recompile Success Rate and Coverage Equivalence Rate.[5]

5.2 Disassembly

In order to combine disassemblers, `DisCo` evaluates the performance of commonly used reverse engineering tools using *Correctly identified function starts (CFS)* as well as *precision and recall* and their derived F1 scores as key metrics for comparison of disassemblers.
Angr outperforms at the disassembly of `MIPS`-binaries, where it has a 4.04% better F1-score than Ghidra. However it performs worse than Ghidra with *ARM*-binaries.

	GCC		**Clang**	
Disassembler	**Average**	**5PWC**	**Average**	**5PWC**
Angr	82.0	71.5	86.8	77.3
Ghidra	76.8	57.2	85.3	70.1

Table 2: Comparison of Angr and Ghidra for GCC and Clang.[11]

	MIPS		**ARM**	
Disassembler	**GCC**	**Clang**	**GCC**	**Clang**
Angr	82.0	86.8	50.2	43.9
Ghidra	76.8	85.3	87.0	91.3

Table 3: Comparison of Angr and Ghidra for MIPS and ARM across GCC and Clang.

Tool	Instruction Boundary				Function Boundary				# Timeout	# Exception	# Segfault
	Precision	Recall	F1 Score	# Invalid	Precision	Recall	F1 Score	# Invalid			
Angr	0.886	0.797	0.830	1	0.404	0.667	0.490	1	16	364	262
Ghidra	0.954	0.828	0.873	0	0.855	0.714	0.766	0	13	0	0

Table 4: Comparison of Angr and Ghidra on Instruction Boundary and Function Boundary metrics.[6]

6 Evaluation

7 Related Work

DisCo: Combining Disassemblers for Improved Performance
The `DisCo` (Disassembler Combination) project combines several industry-grade disassemblers into one novel open-source tool. It can be used to improve other disassemblers. `Ghidra+`, the improved version of Ghidra can achieve a 13,6% better F1 score compared to the original.[11]

DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios
`Decompile Bench` is a large benchmark for evaluating decompiler accuracy and robustness. It gathers empirical security insights via a real-world evaluation framework into an open-source benchmark for decompiler evaluation.
The paper compares 12 decompilers, of which two are decompilation-specialized models and four are general-purpose LLMs.[5]

An Empirical Study on ARM Disassembly Tools
This paper performs the first empirical study of `ARM`-architecture disassembly tools. It evaluates popular benchmarks and real programs.
1040 real-world programs were cross-compiled in addition to 19 benchmark programs into 1896 binaries using multiple obfuscation methods.
The findings of this paper reveil the root causes of failed analysis and provides insights and future directions for improvements.[6]

8 Conclusion

References

[1] Ieee standard for software maintenance. *IEEE Std 1219-1993*, pages 1–45, 1993. doi: 10.1109/IEEESTD.1993.115570.

[2] P-code reference manual. 2023. URL `https://ghidra.re/ghidra_docs/languages/html/pcoderef.html`.

[3] Alexei Bulazel. Working with ghidra's p-code to identify vulnerable function calls. *Infiltrate Security Conference 2019*, May 2019.

[4] Elliot Chikovsky and James H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE*, 1990. URL `https://ieeexplore.ieee.org/document/43044`.

[5] Cui Yuxin Wang Hao Qin Siliang Wang Yuand Bolun Zhan Zhang Chao Gao, Zeyu. DecompileBench: A comprehensive benchmark for evaluating decompilers in real-world scenarios.

In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23250–23267, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1194. URL https://aclanthology.org/2025.findings-acl.1194/.

[6] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 401–414, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397377. URL https://doi.org/10.1145/3395363.3397377.

[7] Chris Eagle Kara Nance. *The Ghidra Book: The Definite Guide*. No Starch Press, 2020.

[8] Ralph Meier. Reverse engineering - introduction to the world of disassembling and decompiling. *scip Labs*, Jan 2021. URL https://www.scip.ch/en/?labs.20211202.

[9] Nico Naus, Freek Verbeck, and Binoy Ravindran. A formal semantics for p-code. *Springer and VSTTE'22*, 2022.

[10] NSA. Sleigh - a language for rapid processor specification, 2023. URL https://fossies.org/linux/ghidra/GhidraDocs/languages/html/sleigh.html.

[11] Sri Shaila, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Disco: Combining disassemblers for improved performance. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 148–161, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390583. doi: 10.1145/3471621.3471851. URL https://doi.org/10.1145/3471621.3471851.

[12] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (state of) the art of war: Offensive techniques in binary analysis. *IEEE Secur. Priv.*, 2016. doi: 10.1109/SP.2016.17.

[13] Michael Sikorski and Andrew Honig. *Practical Malware Analysis The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012. ISBN 1593272901.

[14] subwire and lockshaw. throwing a tantrum, part 1: angr internals. URL https://angr.io/blog/throwing_a_tantrum_part_1/.

[15] Douglas Thain. *Introduction to Compilers and Language Desgin: Second Edition*. University of Notre Dame, 2020.

[16] The angr project. Claripy documentation, . URL https://api.angr.io/projects/claripy/en/latest/index.html. Accessed: 2025-09-01.

[17] The angr project. Core concepts, . URL https://docs.angr.io/en/latest/core-concepts/toplevel.html. Accessed: 2025-09-01.

[18] The angr project. Loading a binary, . URL https://docs.angr.io/en/latest/core-concepts/loading.html#loading-a-binary. Accessed: 2025-09-01.

[19] Jeffery von Ronne, Ning Wang, and Michael Franz. Interpreting programs in static single assignment form. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '04, page 23–30, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581139098. doi: 10.1145/1059579.1059585. URL https://doi.org/10.1145/1059579.1059585.