Exploring Errors in Binary-Level CFG Recovery

Anjali Pare and Prasad A. Kulkarni
Electrical Engineering and Computer Science, University of Kansas, Lawrence, Kansas, U.S.A.

Keywords: Reverse Engineering, Control-Flow Graphs, Disassembly.

Abstract: The control-flow graph (CFG) is a graphical representation of the program and holds information that is critical to the correct application of many other program analysis, performance optimization, and software security algorithms. While CFG generation is an ordinary task for source level tools, like the compiler, the loss of high-level program information makes accurate CFG recovery a challenging issue for binary-level software reverse engineering (SRE) tools. Earlier research shows that while advanced SRE tools can precisely reconstruct most of the CFG for the programs, important gaps and inaccuracies remain that may hamper critical tasks, from vulnerability and malicious code detection to adequately securing software binaries.
In this work, we perform an in-depth analysis of control-flow graphs generated by three popular reverse engineering tools - angr, radare2 and Ghidra. We develop a unique methodology using manual analysis and automated scripting to understand and categorize the CFG errors over a large benchmark set. Of the several interesting observations revealed by this work, one that is particularly unexpected is that most errors in the reconstructed CFGs appear to not be intrinsic limitations of the binary-level algorithms, as currently believed, and may be simply eliminated by more robust implementations. We expect our work to lead to more accurate CFG reconstruction in SRE tools and improved precision for other algorithms that employ CFGs.

1 INTRODUCTION

Software reverse engineering (SRE) is the process of analyzing a software system to extract design and implementation information, especially in cases when the high-level source code is missing (Eilam, 2005). SRE is a challenging task. The (forward) software engineering process that translates a program written in a high-level programming language to a binary executable naturally loses important high-level program information, including data types, function and code layout, and control-flow information. Furthermore, developers may use code *obfuscation*, hand-written assembly code, and other such techniques to intentionally make it even harder to statically recover all aspects of the original code syntax and semantic information (Collberg et al., 1997; Cozzi et al., 2018).

SRE involves several algorithms and heuristics that attempt to recover different aspects of program information from the input binary. Steps include, binary *disassembly* – translating binary code from machine code to assembly language, *decompilation* – translating assembly/machine code to a higher-level language representation, like C/C++, and *binary analysis* – recovering the control-flow and data-flow information of the binary executable. These steps are supported by several finer-level algorithms to perform tasks, such as code discovery or code and data disambiguation, function entry/exit point recovery, function signature recovery, indirect branch target resolution, control-flow graph generation, data type recovery, array start and bound detection, etc.

Many steps in the SRE process are speculative and imprecise (Wartell et al., 2011; Meng and Miller, 2016). Fortunately, it is now well-established that code that is generated by standard compilers, and without the complexities introduced by obfuscation and hand-written assembly, has more predictable properties and clearer patterns (Andriesse et al., 2016; Hawkins et al., 2017). However, even in such simpler cases, complications remain during the SRE process that introduce errors in the program information that is extracted by even the most sophisticated reverse engineering tools (Pang et al., 2021; Andriesse et al., 2016). Such errors can have an out-sized effect on the accuracy of other security and performance transformations that use this information (Caballero and Lin, 2016; Vaidya et al., 2021; Burow et al., 2017).

Measuring the precision of the different stages of the SRE process is important to not only understand

Pare, A. and Kulkarni, P.
Exploring Errors in Binary-Level CFG Recovery.
DOI: 10.5220/0012435400003648

In *Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024)*, pages 546-557
ISBN: 978-989-758-683-5; ISSN: 2184-4356

and resolve their remaining challenges, but also to more clearly present the capabilities of the modern SRE tools to researchers and developers that may then employ them to build more advanced algorithms to improve the security and performance of binary programs. Consequently, researchers have built elaborate frameworks to measure and quantify the accuracy of modern SRE tools for different binary analysis tasks (Pang et al., 2021; Kline. and Kulkarni., 2023).

Unfortunately, while these assessment frameworks and research works report the accuracy numbers of different SRE tasks and conduct light analysis of the results, they typically do not perform a thorough study to deeply understand the issues that cause the observed inaccuracies. Such a detailed study will enable us to measure and categorize the leading causes of the inaccuracy in different SRE processes. We can then further study if and which categories of inaccuracies are intrinsic limitations of the reversing process due to loss of program information, or if they are shortcoming of the adopted algorithm or implementation. Such detailed analysis will help the SRE engineers resolve issues and improve their tools or help researchers focus their efforts on developing better algorithms for specific sub-problems.

An exhaustive study to understand errors in any SRE process is often skipped because it may necessitate a tedious manual study and reasoning over low-level assembly/machine code. Our goal in this work is to develop a systematic mechanism and conduct such studies to examine and understand the sources of errors in key SRE steps. In this current study, we focus our efforts on investigating the causes of imprecision in the important *control-flow graph* (CFG) recovery step during binary disassembly[1].

For this work, we employ the open-source framework built by Pang et al. (Pang et al., 2021) to obtain the *ground-truth* information from the LLVM compiler, and their binary-level scripts to retrieve the CFG from three popular SRE tools, Angr (Shoshitaishvili et al., 2016), Ghidra (NSA, 2023) and Radare2 (Àlvarez, 2023). We leverage this information and develop our own systematic and iterative approach using both manual analysis and automated scripting to uncover and categorize the key causes of imprecision during the CFG recovery by the different SRE tools compared to the ground truth. We limit the scope of our current work to the study of *true negative* CFG edges; in other works, we only attempt to understand why certain CFG edges that are present in the ground truth are *excluded* or *missed* in the CFGs generated for binaries by the SRE tools.

Among the three SRE tools and benchmark sets we employ for this work, we uncover 14 categorizes of instruction and block patterns that account for most of the *true negative* CFG edges. Surprisingly, we find that most causes of imprecise CFG generation by SRE tools are not intrinsic limitations imposed by the binary format, but seem to be algorithmic errors that could be fixed by more robust tool implementation. Researchers and tool engineers can use our results to reduce the number of true negative CFG edges and improve the precision of SRE tools.

We make the following contributions in this work.

1. We develop a novel semi-automatic method to study and understand the causes of imprecision in various stages of the SRE process.
2. We apply our methodology to study and understand the causes of *true negatives* during binary-level CFG recovery.
3. We categorize instruction and block patterns that produce most true negative edges in the CFGs generated by modern binary-level SRE tools.
4. We collect and analyze our data about the precision of CFG recovery by three modern SRE tools collected over a large benchmark set with multiple compilers, and report many interesting and some counter-intuitive observations.

2 RELATED WORK

Several researchers have reported the causes of imprecision during different steps of binary disassembly. In this section we present related works and contrast them with the work we do in this paper.

Binary disassembly is difficult due to the challenges imposed by separating code from data, padding bytes inserted by the compiler for performance improvement, variable-sized instructions, indirect control transfer instructions, etc. In fact, it has been shown that solving the disassembly problem is equivalent to the Halting Problem and is therefore unsolvable in general (Horspool and Marovac, 1980). Meng and Miller found example code constructs in standard libraries that were hard to analyze for many disassemblers (Meng and Miller, 2016). Andriesse et al. later showed that although such hard-to-analyze code constructs are possible, they are not very common in most unobfuscated binaries that are compiled by standard compilers (Andriesse et al., 2016).

Pang et al. studied the accuracy of different disassembly stages in 9 open-source disassemblers (Pang

[1] A CFG has a node for every basic block and an edge for every possible control flow transfer in the function. CFGs are used in static analysis and compiler applications because of their ability to represent the flow of a program.

et al., 2021). They also developed an experimental methodology and framework to conduct this study. They explored the algorithms and heuristics used during the disassembly process by each studied tool, and identified some of their weaknesses and benefits. Since different SRE tools use distinct reversing algorithms, their precision and effectiveness varies for each task. Researchers have attempted to account for this reality by developing a method to combine the analysis results from multiple disassembler to achieve a better overall result (Shaila et al., 2021).

In contrast, our goal in this work is to determine and categorize the causes for imprecision in CFGs reconstructed by popular open-source disassemblers. We employ the framework built and open-sourced by Pang et al. for this work (Pang et al., 2021).

3 METHODOLOGY

In this section, we discuss the experimental setup and methodology used to study CFG reconstruction in reverse engineering tools, and analyze and compare the performance of the tools.

Our unique methodology uses manual analysis and automated scripting to understand and categorize the CFG errors over a large benchmark set. We also study the true positive, false positive and false negative edges of the CFGs to analyze the overall performance of the tools. The main steps of our methodology, including the tool analysis and CFG edge categorization steps, are illustrated in Figure 1. We explain our methodology below.

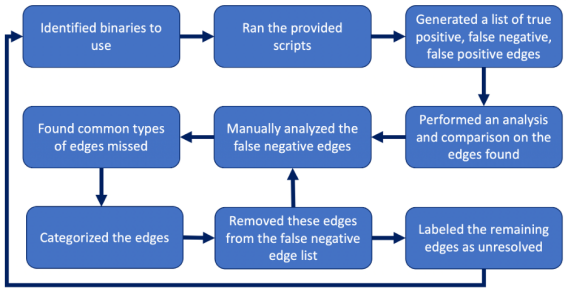

Figure 1: Methodology.

First, we determined the benchmarks and tools to use for this work. We employ binaries from a subset of the benchmarks provided by the earlier SoK work(Pang et al., 2021). More specifically, we use binaries from 5 categories of benchmarks: (a) 13 programs from the *client* set, (b) 104 programs from *coreutils* (c) 3 programs from *findutils*, (d) 15 programs from *binutils*, and (e) 1 *servers* benchmark. All binaries were built for the x64-Linux platform using GCC and Clang compilers with optimization levels of O0 and O3. We use these binaries to study the accuracy of control-flow graphs generated by three popular binary analysis tools, angr (version 9.2.6), radare2 (version 5.7.9) and Ghidra (version 10.1.2).

We use the scripts and ground truth information provided by the SoK work (Pang et al., 2021) to compare the accuracy of the control-flow graphs generated by each tool for our benchmark configurations. For each case, we generate a list of true positive, false positive and false negative edges in the respective CFG, compared to the ground truth. In this work we focus on studying and categorizing only the *true* CFG edges that are missed by the selected SRE tools. These edges are called the *false negative* edges.

Next we employ manual analysis and automated Ghidra scripting to further analyze and categorize the false negative edges. We categorize the CFG edges missed into 14 categories, as discussed in Section 4.

Additionally, we explore the following questions:
(1) What percentage of the CFG edges are correctly found by the tools for all the benchmarks?
(2) What percentage of true positive edges are found by all the tools combined for all the benchmarks?
(3) What fraction of edges are missed by each of the tools as compared to the ground truth?
(4) What fraction of edges are missed by all the tools combined?
(5) How do the categories of missed edges differ for the different reverse engineering tools, compilers, optimization levels, and benchmarks?
(6) What are the most common categories of missed edges for the different reverse engineering tools, compilers, and optimization levels?
(7) How many additional edges (false positives) are found?
(8) What fraction of false negative edges are missed by pairs of tools?
(9) What fraction of common edges are found by pairs of tools?

Finally, we determine a set of categories of false negative edges that we believe are not caused by an intrinsic algorithmic or analysis limitation, and hence can be easily correctly by better software implementations. For our final *post-processing* step, we remove this set of *spurious* false negative edges from the list of missed edges to compute the true accuracy of the tools regarding precise CFG generation. This comparison helps us study the benefit of *fixing* the spurious false negative CFG edge categories to improve CFG precision for algorithms that require CFGs.

4 CATEGORIES OF FALSE NEGATIVE EDGES

We use our methodology described in the last section to find and analyze the most prevalent issues that result in *false negative* CFG edges for the three binary analysis tools studied in this work. We use the observations from this analysis to categorize the false negative CFG edges. We describe these causes and categories of false negative CFG edges in this section.

Edges Across Functions. The binary analysis tools sometimes miss correctly detecting edges that start at a basic block in one function and end at a basic block in another function. We discover these edges by analyzing the function entry points found by the respective tools. Figure 2 displays an example of this case with an edge 0x404ba8 → 0x404bda that is missed.

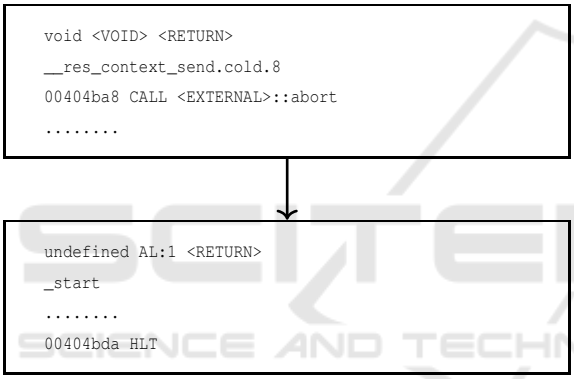

Figure 2: Edges missed across functions.

Loop Edges. Sometimes edges from a basic block to itself are missed. Figure 4 displays an example of this case with an edge 0x487858 → 0x487858.

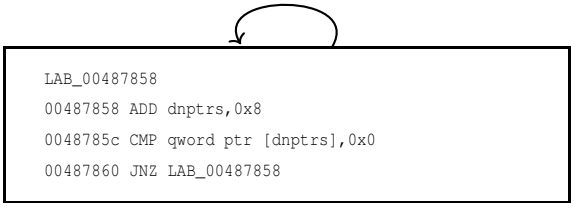

Figure 3: Same edges missed.

Edges Across Jumps. Some edges from a basic block that end in JMP instruction to the target basic block are also missed. Figure 3 displays an example of this case with an edge 0x415415 → 0x415503.

NOP Edges. Some edges from a basic block that end in a NOP instruction to the following block are missed. Figure 5 displays an example of this case with an edge 0x4072ba → 0x4072bb that is missed.

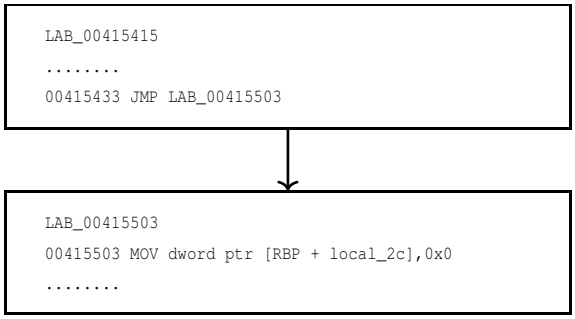

Figure 4: Jump edges missed.

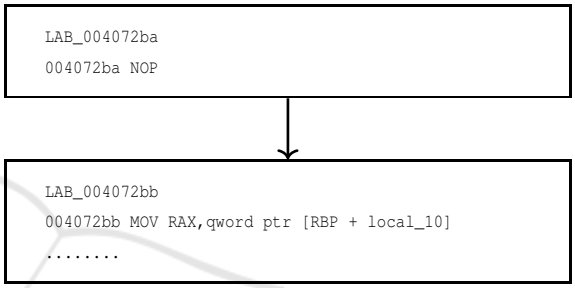

Figure 5: Nop edges missed.

Conditional Edges. Sometimes the *taken* or the *fall-through* edges from a block that ends in a conditional jump (such as JZ) are missed. Figures 6 and Figure 7 display examples of this case with the fall-through edge 0x48159a → 0x48175b, or the taken edge 0x48159a → 0x481760 missed, respectively.

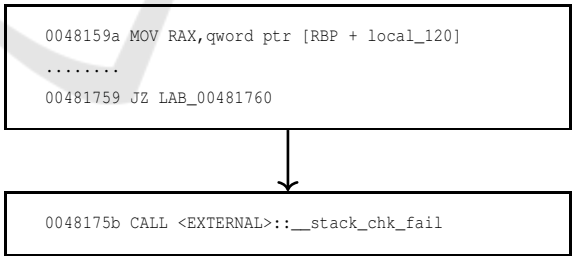

Figure 6: Conditional edges missed 1.

Indirect Call Edges. Sometimes edges from a basic block that end in an indirect call to the following block miss detection by the tools. Figure 8 displays an example of this case with an edge 0x4324af → 0x4324cc that is missed.

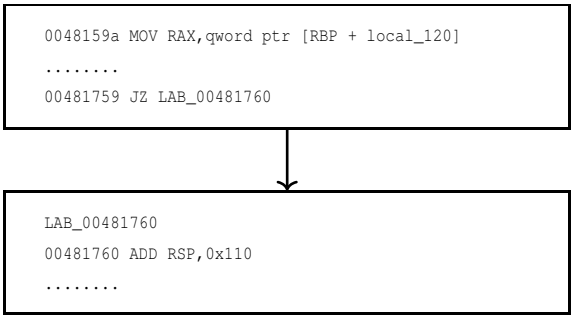

Figure 7: Conditional edges missed 2.

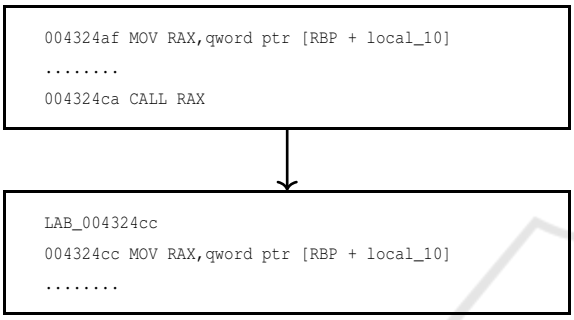

Figure 8: Indirect Call edges missed.

Across Block Edges. There are cases when the SRE tools end basic blocks at non-terminator instructions (instructions like a JMP or a JNZ are terminators). This situation happens for large basic blocks. The ground truth (from LLVM) does not perform the block split. For example, the ground truth may have a single edge from 0x455c3d → 0x455f15. For large basic blocks, the tools may split the basic blocks and find multiple edges (for instance, 0x455c3d → 0x455d42, 0x455d42 → 0x455d8a, 0x455d8a → 0x455f15), but not the edge in the ground truth. Then, the edge in the ground truth shows up as a false negative (while edges from the split basic blocks are detected as false positives).

Undefined Function Edges. There are some cases where the tools are unable to determine the function starting address, which leads to undefined function edges. In Ghidra's decompiler window, these functions are labeled as UndefinedFunction. Figure 9 displays an example of this case with a false negative edge 0x403658 → 0x40368a

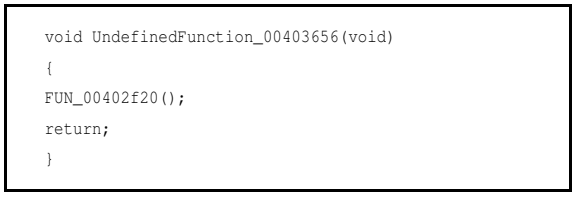

Figure 9: Undefined function edges missed.

MOV Edges. Some edges from a basic block that end in a MOV or a MOVZX instruction to the following block also go undetected. Figure 10 displays an example of this case with an edge 0x406269 → 0x40626d that is missed by some tools.

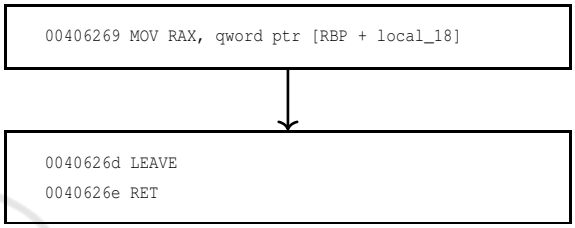

Figure 10: Mov edges missed.

Call Edges. There are some cases where the tools miss edges that start at a block that terminates with a CALL instruction to a function and ends at a block following the CALL instruction. Figure 11 displays an example of this case with an edge 0x44eccb → 0x44ece9 that is missed.

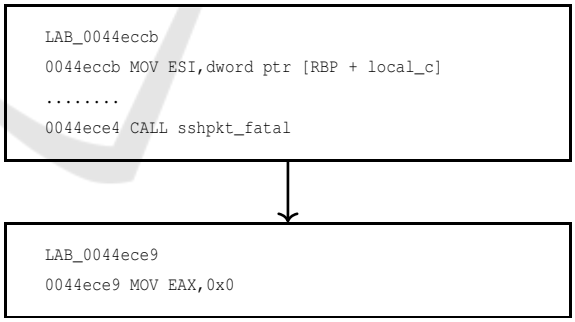

Figure 11: Call edges missed.

Add Edges: These are missed edges from a basic block that end in an ADD instruction to the following block. Figure 12 displays an example of this case with an edge 0x404657 → 0x40465f that is missed.

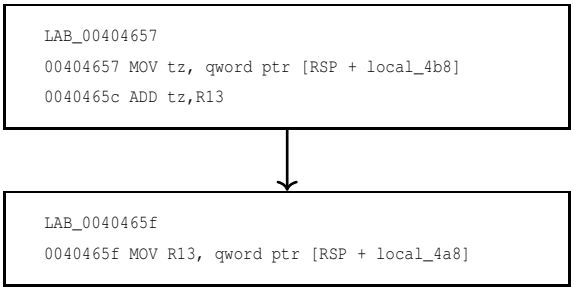

Figure 12: Add edges missed.

Code Mislabeled Data. There are some cases where the SRE tools find it difficult to distinguish the code and data sections of the code. This leads to missed edges in these sections.

Add Jump Edges. Some CFG edges from a basic block that end in an ADD instruction to the following block that starts with a JMP instruction were also missed. Figure 13 displays an example of this case with an edge 0x459df4 → 0x459e06 that is missed

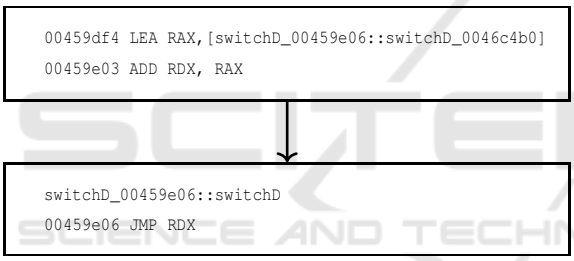

Figure 13: Add Jump edges missed.

Unresolved Edges. There are a few false negative CFG edges that we could not match with a common pattern. We leave such edges as unresolved edges.

5 RESULTS

We study the true positive, false negative and false positive edges of a control-flow graphs to analyze the performance of the tools in CFG generation. We report our results and observations in this section.

5.1 True Positive Edges

Table 1 provides information about the percentage of edges found by the tools for GCC and Clang with optimization levels of O0 and O3. As seen in the table, all the tools are able to find most of the edges in the ground truth. It is observed that all the tools perform the same or better in most cases when compiled with GCC_O0 as opposed to GCC_O3. angr displays a similar pattern for binaries compiled with Clang as well. No such consistent pattern was observed with radare2 or Ghidra for Clang. Next, angr and radare2 performed comparatively better than Ghidra for all the benchmarks. In addition, performance of the tools was better when compiled with GCC as compared to Clang for optimization level O0.

Table 2 provides information about the percentage of edges found by all the tools combined as compared to the ground truth. Similar to the results seen in Table 1, performance of the tools was better when compiled with GCC as compared to Clang for both optimization levels, O0 and O3.

5.2 Edges Missed by the Tools

Table 3 provides information about the total number of edges missed individually and the common edges missed by the tools as compared to the ground truth. As seen in the table, when the total number of edges missed across all the benchmarks is calculated, angr has the least and Ghidra has the highest number of false negative edges. Furthermore, the number of edges missed when all three tools are used, is significantly lower compared to their individual performance for all the benchmarks. In addition, when the overall performance of the compilers is observed across all the benchmarks and tools, we see that binaries compiled with GCC generate a lesser number of false negative edges as compared to those compiled with Clang, which implies better performance. This observation is similar to that seen in Table 2.

5.3 Types of Edges Missed by the Tools

As mentioned in section 4, we categorized the edges missed into 14 types. For more efficient analysis, we combine the ADD, ADD JMP, and MOV edge categories into a Fall Through edge category.

As seen in Table 3, the number of edges missed by the combination of all the tools is lower compared to their individual performance. These edges are categorized in Table 4. We find that NOP, Undefined Function and Call edges have the highest number of edges. angr misses a higher number of NOP, Call and Code Mislabeled Data edges. radare2 and Ghidra miss NOP, Undefined Function, and Call edges, most commonly. In addition, binaries compiled with GCC_O3 have the highest number of missed NOP edges followed by Call and Undefined function edges. A similar behavior was observed for binaries compiled with Clang_O3. For binaries compiled with Clang_O0, Jump, NOP and Undefined Function edges were most

Table 1: Percentage of edges found by the tools for GCC_O0, GCC_O3, Clang_O0 and Clang_O3.

Files	Tools	GCC_O0	GCC_O3	Clang_O0	Clang_O3
Coreutils	angr	0.9983	0.9746	0.9971	0.9824
	radare2	0.9805	0.9776	0.9688	0.9888
	Ghidra	0.9660	0.9578	0.9200	0.8599
Findutils	angr	0.9994	0.9821	0.9974	0.9872
	radare2	0.9974	0.9695	0.9786	0.9919
	Ghidra	0.9757	0.9698	0.8661	0.9427
Binutils	angr	0.9936	0.9706	0.9642	0.9822
	radare2	0.9463	0.9866	0.9779	0.9526
	Ghidra	0.9336	0.9765	0.8433	0.9091
Clients	angr	0.9962	0.9758	0.9920	0.9826
	radare2	0.9932	0.9871	0.9897	0.9925
	Ghidra	0.9734	0.9684	0.9669	0.9767
Servers	angr	-	-	0.9941	0.9857
	radare2	-	-	0.9993	0.9957
	Ghidra	-	-	0.9389	0.9684

Table 2: Percentage of combined edges found by the tools for GCC_O0, GCC_O3, Clang_O0 and Clang_O3.

Files	GCC_O0	GCC_O3	Clang_O0	Clang_O3
Coreutils	0.95	0.92	0.89	0.84
Findutils	0.97	0.93	0.85	0.93
Binutils	0.90	0.94	0.82	0.88
Clients	0.97	0.96	0.96	0.96
Servers	-	-	0.94	0.96

commonly missed.

As seen in Table 4, there are some edges that are categorized differently by different tools. For instance Ghidra labels 26 edges as missed Across Block edges for GCC_O3. However, these are categorized as Fall through edges by angr and radare2. A similar case is observed for Clang_O0. For Clang_O3, the 8 edges found by Ghidra in the Across Block category are labeled as unresolved by angr and radare2. Another such instance is the Undefined Function and Code Mislabeled as Data edges. angr labels most of these edges as Code Mislabeled as Data whereas radare2 and Ghidra label them as Undefined Function edges.

Table 5 and Table 6 provide information about the types of false negative edges for different reverse engineering tools, different compilers, different optimization levels, and different types of benchmarks. Based on the table, Undefined Function, Code Mislabeled as Data, Across Block, Conditional, and Jump edge categories have the highest number of edges.

angr misses a higher number of Across Block, Conditional, Jump, and Fall through edges. radare2 misses a higher number of Conditional, Undefined Function, Fall Through and Jump edges. Ghidra misses Across Block, Jump, Undefined Function and Code Mislabeled as Data edges, most commonly. Compared to the other tools, Ghidra misses the most cases of Across Block and Code Mislabeled as Data edges, whereas angr misses the most cases of Indirect call and Loop edges. radare2 misses the most cases of Conditional and Fall through edges as compared to the other tools. Furthermore, when compiled with optimization level of O3, the number of NOP edges missed by the tools increases considerably.

5.4 True Positive Edges after Post Processing

Apart from the Across function, Indirect Call, Undefined Function and Code Mislabeled as Data Edges, all the other category edges can be added during the post processing analysis of the CFGs. Once these edges are added and are no longer considered as false negative edges, we compare the new performance of the tools to the ground truth. Table 7 provides information about the percentage of edges found by the

Table 3: Number of edges missed by the tools. GT, A, R, G stand for ground truth, angr, radare2 and Ghidra, respectively.

Files	Compiler	GT	A	R	G	All	A %	R %	G %	All %
Coreutils	GCC_O0	199037	342	3881	6771	0	0.17	1.95	3.40	0.00
	GCC_O3	216305	5498	4841	9131	77	2.54	2.24	4.22	0.04
	Clang_O0	248508	716	7744	19877	11	0.29	3.12	8.00	0.00
	Clang_O3	214830	3777	2397	30106	49	1.76	1.12	14.01	0.02
Findutils	GCC_O0	26049	16	69	633	0	0.06	0.26	2.43	0.00
	GCC_O3	31497	565	960	951	11	1.79	3.05	3.02	0.03
	Clang_O0	39728	104	852	5318	0	0.26	2.14	13.39	0.00
	Clang_O3	30320	389	246	1736	14	1.28	0.81	5.73	0.05
Binutils	GCC_O0	118531	759	6369	7867	0	0.64	5.37	6.64	0.00
	GCC_O3	116809	3435	1565	2747	108	2.94	1.34	2.35	0.09
	Clang_O0	125622	4498	2775	19683	4	3.58	2.21	15.67	0.00
	Clang_O3	125514	2240	5954	11407	78	1.78	4.74	9.09	0.06
Clients	GCC_O0	199649	756	1367	5319	2	0.38	0.68	2.66	0.00
	GCC_O3	245260	5937	3154	7762	506	2.42	1.29	3.16	0.21
	Clang_O0	257032	2047	2659	8512	93	0.80	1.03	3.31	0.04
	Clang_O3	257848	4483	1922	6002	179	1.74	0.75	2.33	0.07
Servers	Clang_O0	14293	84	10	874	0	0.59	0.07	6.11	0.00
	Clang_O3	14675	210	63	463	5	1.43	0.43	3.16	0.03
Total		2481507	35856	46828	145159	1137	1.44	1.89	5.85	0.05

Table 4: Types of false negative edges missed by all the tools across all the benchmarks. A, R, G stand for angr, radare2 and Ghidra, respectively.

Compiler	GCC_O0			GCC_O3			Clang_O0			Clang_O3		
Tool	A	R	G	A	R	G	A	R	G	A	R	G
Across Blocks	0	0	0	0	0	26	0	0	13	0	0	8
Across Func	2	2	2	0	0	0	0	0	0	0	0	0
Conditional	0	0	0	18	18	18	0	0	0	0	0	0
Jump	0	0	0	6	6	6	25	25	25	12	12	12
Loop	0	0	0	0	0	0	14	14	14	5	5	5
Call	0	0	0	98	99	99	1	1	1	2	2	2
Nop	0	0	0	470	470	470	16	16	16	245	245	245
Undef Func	0	0	0	3	69	69	6	33	33	50	50	50
Fall Through	0	0	0	31	31	6	13	13	0	1	1	1
Code Data	0	0	0	72	5	5	33	6	6	2	2	2
Unresolved	0	0	0	4	4	3	0	0	0	8	8	0
Total	2	2	2	702	702	702	108	108	108	325	325	325

tools for GCC and Clang with optimization levels of O0 and O3 after post processing. Similar to the data presented in Table 1, all the tools are able to find most of the edges in the ground truth. However, after post processing, it is observed that the performance of the tools improved for all the benchmarks. This comparison helps us study the impact of our work in generating more accurate CFGs that further lead to improved precision for other algorithms that require CFGs.

5.5 Across Function Edges

We use manual analysis and scripting to determine the Across Function edges present in the ground truth. Since these edges start at a basic block in one function and end at a basic block in another function, they are

Table 5: Types of false negative edges across all the benchmarks. A, R, G stand for angr, radare2 and Ghidra, respectively.

Compiler	GCC_O0			GCC_O3			Clang_O0			Clang_O3		
Tool	A	R	G	A	R	G	A	R	G	A	R	G
Across Blocks	745	140	18699	5974	160	15971	224	61	11263	527	88	8150
Across Func	2	3	2	0	0	2	2	4	1	0	0	0
Conditional	342	1360	158	4422	3614	27	955	4999	2	5764	1778	1033
Jump	116	4439	232	3517	3588	217	2072	3970	6293	3131	5076	3974
Loop	0	0	0	23	0	4	33	14	20	309	5	315
Call	326	22	19	212	262	522	159	45	75	293	87	341
Indirect	226	1	2	156	0	1	279	0	1	239	0	1
Nop	4	142	4	507	499	471	16	17	16	250	274	260
Undef Func	0	1406	1211	138	1830	2304	48	1683	1702	117	57	3143
Fall Through	111	3981	48	403	524	17	3569	2686	16	386	1647	74
Code Data	0	185	214	77	9	1031	80	563	34845	36	1542	32113
Unresolved	1	7	1	6	34	24	12	0	30	47	28	310

incorrect and should not be present in the CFG. We find 5 such cases in the ground truth. When we observe the Across Function edges missed by the tools in Table 5, we find that angr, radare2 and Ghidra miss 4, 7 and 5 such edges, respectively.

5.6 True Positive Edges after Post Processing

Table 7 provides information about the percentage of edges found by the tools for GCC and Clang with optimization levels of O0 and O3 after post processing. Similar to the data presented in Table 1, all the tools are able to find most of the edges in the ground truth. However, after post processing, it is observed that the performance of the tools improved for all the benchmarks.

6 DISCUSSION

6.1 Edges Missed by the Tools

As seen in the Table 3, edges missed by the combination of all the tools was lower than the edges missed individually. This would imply that utilizing all the SRE tools leads to lower false negative edges. For most of the missed edges, there was at least one tool that was able to find it accurately and lead to better CFG reconstruction.

As previously mentioned, there are some false negative edge cases that could be easily eliminated. These categories include Conditional, Jump, Loop, Call, NOP, Fall through and Across Block edges. It is unclear why the algorithms that the tools use lead to these categories of false negative edges. However, because of the nature of these edges, they may be eliminated during post processing. Once the edges are removed, we calculate the performance of the tools as shown in Table 7. After post processing, the performance of the tools significantly improved which shows that with more robust implementations, we will be able to generate more accurate CFGs.

6.2 Types of Edges Missed by the Tools

As seen in the previous section, some of the false negative edges missed are Code Mislabeled as Data edges. It is hard for reverse engineering tools to separate code and data regions. Therefore, we see a high number of edges mislabeled as data leading to false negative edges.

With the help of Ghidra's listing view, we are able to view bytes that Ghidra was able to disassemble but not assign to a particular function. Ghidra labels these functions as UndefinedFunctions. Such functions may be created by the SRE tools when they are unable to determine the start address of the function. Because of this, these edges are not analyzed and lead to false negatives. To reduce the false negative edges, the tools would need to accurately determine the function start and end addresses.

Another category with a high number of false negative edges is the Across Blocks edge category. angr researchers informed us that these edges are missed because of a limitation that angr inherits from libvex. This limitation states that basic blocks can only contain certain number of instructions after which the blocks are terminated. This leads to the Across Block

Table 6: Types of false negative edges.

Files	Compiler	Blocks	Func	Cond	Jump	Loop	Call	Indirect	Nop	Undef Func	Fall-Through	Code Data	Unresolved
Clients	GCC_O0	320	2	217	30	0	73	110	0	0	3	0	1
	GCC_O3	2124	0	2315	625	22	138	98	338	39	156	77	5
angr	Clang_O0	190	1	927	393	33	66	161	16	48	170	32	10
	Clang_O3	318	0	2965	652	53	11	145	159	36	116	1	27
Clients	GCC_O0	21	2	85	34	0	6	0	2	1210	6	0	1
	GCC_O3	18	0	118	707	0	101	0	305	1761	133	4	7
radare2	Clang_O0	10	2	191	475	14	18	0	17	1667	65	200	0
	Clang_O3	8	0	834	698	0	66	0	169	26	84	20	17
Clients	GCC_O0	4078	2	0	0	0	4	0	0	1205	2	28	0
	GCC_O3	4337	0	25	84	3	122	0	305	2129	17	729	11
Ghidra	Clang_O0	1727	1	2	1003	20	44	0	16	1702	9	3967	21
	Clang_O3	1166	0	900	345	55	37	0	162	874	66	2374	23
Coreutils	GCC_O0	106	0	112	47	0	10	67	0	0	0	0	0
	GCC_O3	1967	0	1197	2142	0	27	44	70	51	0	0	0
angr	Clang_O0	28	0	6	541	0	10	60	0	0	71	0	0
	Clang_O3	135	0	1474	1817	193	23	48	37	13	36	1	0
Coreutils	GCC_O0	105	1	987	1374	0	9	1	133	174	1091	0	6
	GCC_O3	123	0	2070	2172	0	40	0	89	59	262	4	22
radare2	Clang_O0	42	0	3197	2089	0	20	0	0	12	2332	52	0
	Clang_O3	66	0	440	1740	1	13	0	43	13	39	42	0
Coreutils	GCC_O0	6755	0	0	0	0	9	2	0	4	0	1	0
	GCC_O3	8289	0	0	111	0	327	1	71	151	0	173	8
Ghidra	Clang_O0	4469	0	0	1875	0	13	1	0	0	0	13519	0
	Clang_O3	2186	0	115	2751	196	229	0	44	1943	3	22397	242
Findutils	GCC_O0	6	0	2	4	0	0	4	0	0	0	0	0
	GCC_O3	234	0	166	108	1	0	3	11	14	28	0	0
angr	Clang_O0	1	0	2	97	0	0	4	0	0	0	0	0
	Clang_O3	43	0	231	74	14	0	3	3	10	6	5	0
Findutils	GCC_O0	1	0	2	32	0	0	0	3	0	31	0	0
	GCC_O3	3	0	752	71	0	120	0	12	0	1	1	0
radare2	Clang_O0	2	0	394	273	0	4	0	0	4	173	2	0
	Clang_O3	1	0	120	93	0	1	0	9	10	11	1	0
Findutils	GCC_O0	633	0	0	0	0	0	0	0	0	0	0	0
	GCC_O3	714	0	2	7	1	73	0	11	9	0	129	5
Ghidra	Clang_O0	218	0	0	414	0	17	0	0	0	0	4668	1
	Clang_O3	314	0	6	144	14	74	0	3	23	1	1144	13
Binutils	GCC_O0	313	0	11	35	0	243	45	4	0	108	0	0
	GCC_O3	1649	0	744	642	0	47	11	88	34	219	0	1
angr	Clang_O0	1	1	16	1032	0	81	29	0	0	3292	44	2
	Clang_O3	23	0	989	532	49	253	25	48	56	216	29	20
Binutils	GCC_O0	13	0	286	2999	0	7	0	4	22	2853	185	0
	GCC_O3	16	0	674	638	0	1	0	93	10	128	0	5
radare2	Clang_O0	5	0	1215	1130	0	3	0	0	0	113	309	0
	Clang_O3	11	0	384	2489	4	7	0	50	6	1513	1479	11
Binutils	GCC_O0	7233	0	158	232	0	6	0	4	2	46	185	1
	GCC_O3	2631	2	0	15	0	0	0	84	15	0	0	0
Ghidra	Clang_O0	4722	0	0	2899	0	1	0	0	0	6	12048	7
	Clang_O3	4388	0	12	718	49	1	1	48	215	4	5941	30
Servers	Clang_O0	4	0	4	9	0	2	25	0	0	36	4	0
angr	Clang_O3	8	0	105	56	0	6	18	3	2	12	0	0
Servers	Clang_O0	2	2	0	3	0	0	0	0	0	3	0	0
radare2	Clang_O3	2	0	0	56	0	0	0	3	2	0	0	0
Servers	Clang_O0	127	0	0	102	0	0	0	0	0	1	643	1
Ghidra	Clang_O3	96	0	0	16	1	0	0	3	88	0	257	2
Total		62002	16	24454	36625	723	2363	906	2460	13639	13462	70695	500

edge category edges in angr. A similar pattern is observed in Ghidra and radare2 when the basic blocks contain a higher number of instructions.

Fall-through edges were another category of edges that were commonly missed by the tools. One reason for this could be that the tools are unable to determine if the instructions in the fall-through blocks are reachable. In addition, the tools could potentially miss call edges if they are unable to determine whether a function returns or not. A call to a non-returning function will not return to the call site and this would lead to false negative edges.

Indirect Call edges were another category of false negative edges discussed in this study. Indirect branch

Table 7: Percentage of edges found by the tools for GCC_O0, GCC_O3, Clang_O0 and Clang_O3 after post processing.

Files	Tools	GCC_O0	GCC_O3	Clang_O0	Clang_O3
Coreutils	angr	0.9997	0.9996	0.9998	0.9997
	radare2	0.9991	0.9996	0.9997	0.9997
	Ghidra	1.0000	0.9985	0.9456	0.8856
Findutils	angr	0.9998	0.9995	0.9999	0.9994
	radare2	1.0000	0.9999	0.9998	0.9996
	Ghidra	1.0000	0.9955	0.8825	0.9611
Binutils	angr	0.9996	0.9996	0.9994	0.9990
	radare2	0.9983	0.9999	0.9975	0.9881
	Ghidra	0.9984	0.9999	0.9040	0.9507
Clients	angr	0.9994	0.9991	0.9990	0.9992
	radare2	0.9939	0.9928	0.9927	0.9998
	Ghidra	0.9938	0.9883	0.9779	0.9873
Servers	angr	-	-	0.9980	0.9986
	radare2	-	-	0.9999	0.9999
	Ghidra	-	-	0.9549	0.9764

targets depend on values that are computed at runtime. Therefore, predicting these statically may be challenging. As seen previously, if the tools are unable to determine whether the indirect call target is a returning or non-returning function they could potentially miss these edges.

Table 5 displays the types of edges missed by the tools. Here it was observed that when compiled with higher optimization levels, the number of NOP edges missed increases. This may be observed because with higher optimization levels, NOP instructions need to be added for code alignment in the basic blocks.

As seen in Table 4 and 5, tools may categorize certain edges differently. The tools categorize the false negative edges based on the control-flow edges found by them. For example, if a tool does not correctly identify a particular function start address, it would categorize that edge as an Undefined Function edge. If the other tools were able to find the function start address correctly, but the edge was still missed, it would be categorized as a different false negative edge type.

6.3 Across Function Edges

We find that the ground truth finds 5 Across Function edges across all the benchmarks. angr, radare2 and Ghidra miss 4, 7 and 5 such edges, respectively. As mentioned previously, these edges are categorized as Across Function edges based on the function entry points found by the tools. radare2 may miss some function entry points which leads to additional across function edges missed by the tool.

6.4 Compiler Optimization

We did not find significant or consistent differences in the performance of the tools when optimization levels of O0 and O3 were used. We hypothesize that since most optimizations that the compiler applies are intra-block and keep the structure of the block intact, we do not see a significant decline or improvement in the performance of the tools when different compiler optimization levels are used. Although, when overall performance was observed, the tools generated more accurate CFGs when binaries were compiled with GCC as opposed to Clang. For this reason, we hypothesize that the tools are configured to perform better on binaries compiled with GCC.

7 FUTURE WORK

There are multiple avenues for future work. Firstly, in this work we only study and categorize the false negative edges of the CFG generated by mainstream binary analyzers. In the future we will perform a similar analysis on the false positive edges of the CFGs. Secondly, we observe that the edges missed by all the tools combined is lower than the edges missed by individual tools. We plan to use this observation to develop techniques that could utilize multiple tools leads to lower the overall false negative edges. Thirdly, we will extend these results will new benchmarks, compilers and binary analysis tools. Finally, one of our main contributions of this work is the observation that many categories of false negative edges

are not intrinsic limitations of the analysis algorithms, but seem to be caused by spurious implementation oversights. Therefore, an important future project is to attempt to fix such mistakes to directly resolve many causes of erroneous CFG edges generated by these tools.

8 CONCLUSION

In this paper we studied three reverse engineering tools angr, radare2 and Ghidra. We used the benchmarks and framework developed in the SoK Binary Disassembly paper (Pang et al., 2021) to compare our results and perform an in-depth analysis of control-flow graphs generated by the tools. We focused on binaries for the x64 architecture, built for Linux operating systems using GCC and Clang compilers, with optimization levels of O0 and O3. With the help of this paper, we studied the performance of the tools for different compilers and optimization levels, found the true positive and false negative edges, categorized the false negative edges, and compared the performance of the tools before and after post processing. By using manual analysis and scripting, we were able to identify that most errors in the generation of CFGs are not because of the limitations of the algorithms. Therefore, they can be easily eliminated by more robust implementations. With the help of this paper, we aimed to provide information about CFG reconstruction errors that may lead to more accurate CFG generation in the SRE tools and any other algorithms that use CFGs.

REFERENCES

Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., and Bos, H. (2016). An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 583–600, USA. USENIX Association.

Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. (2017). Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1).

Caballero, J. and Lin, Z. (2016). Type inference on executables. *ACM Comput. Surv.*, 48(4).

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand.

Cozzi, E., Graziano, M., Fratantonio, Y., and Balzarotti, D. (2018). Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*, pages 161–175. IEEE.

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, Inc., USA.

Hawkins, W., Hiser, J. D., Nguyen-Tuong, A., Co, M., and Davidson, J. W. (2017). Securing binary code. *IEEE Security & Privacy*, 15(6):77–81.

Horspool, R. N. and Marovac, N. (1980). An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229.

Kline., J. and Kulkarni., P. (2023). A framework for assessing decompiler inference accuracy of source-level program constructs. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy - ICISSP,*, pages 228–239. INSTICC, SciTePress.

Meng, X. and Miller, B. P. (2016). Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 24–35, New York, NY, USA. Association for Computing Machinery.

NSA (2023). Ghidra. https://ghidra-sre.org/.

Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., and Xu, J. (2021). Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851.

Shaila, S., Darki, A., Faloutsos, M., Abu-Ghazaleh, N., and Sridharan, M. (2021). Disco: Combining disassemblers for improved performance. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 148–161.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

Vaidya, R., Kulkarni, P. A., and Jantz, M. R. (2021). Explore capabilities and effectiveness of reverse engineering tools to provide memory safety for binary programs. In Deng, R., Bao, F., Wang, G., Shen, J., Ryan, M., Meng, W., and Wang, D., editors, *Information Security Practice and Experience*, pages 11–31, Cham. Springer International Publishing.

Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., and Thuraisingham, B. (2011). Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III 22*, pages 522–536. Springer.

Àlvarez, S. (2023). The official radare2 book. https://book.rada.re/index.html.