Compiler Help for Binary Manipulation Tools

Tugrul Ince and Jeffrey K. Hollingsworth

Computer Science Department University of Maryland College Park, MD 20742
{tugrul,hollings}@cs.umd.edu

Abstract. Parsing machine code is the first step for most analyses performed on binary files. These analyses build control flow graphs (CFGs). In this work we propose a compilation mechanism that augments binary files with information about where each basic block is located and how they are connected to each other. This information makes it unnecessary to analyze most instructions in a binary during the initial CFG build process. As a result, these binary analysis tools experience dramatically increased parsing speeds - 3.8x on average.

1 Introduction

Binary analysis is a common operation for performance modeling [16], computer security [18], maintenance [5], and binary optimization [11]. Each of these tasks requires parsing the executable file to identify functions, data segments, and their interaction with each other. However, parsing executables is not a straightforward task and it is painfully slow since it usually requires decoding every single instruction in the binary. At the higher level, even distinguishing code and data is difficult since they are often stored in adjacent memory. All the information about functions and data locations is actually known during various stages of compilation. However, only some of this information is stored in the binary in the form of symbols. Binary analysis tools that operate on these executables have to regenerate the information that is thrown away by the compiler.

In this work, we propose a novel compilation mechanism that stores useful information about the layout of executable files in tables inside executable files. These tables enable identification of basic blocks and provide support for reconstruction of edges between them. Binary analysis tools, including those that aid development of high performance applications, can parse executables faster and more reliably using these tables. We measured a speed-up in parsing up to 4.4x with an average speed-up of 3.8x. Since these tables are stored in a section that is not loaded into the memory during execution, the memory footprint of executables do not change. Running times of these executables also remain unchanged since we do not in any way modify the execution. The overhead in the compilation time and the increase in file size is manageable - both at around 23%.

2 Difficulties of Binary Parsing

The process of analyzing bytes from a file and generating abstractions like instructions, functions and CFGs is called binary parsing. It is a tedious task with many challenges, such as distinguishing code from data. Since both code and data are stored the same way, there is really no easy way of identifying whether a sequence of bytes correspond to code or data. Current parsing techniques use hints to identify code and mark the remaining bytes as data. These hints usually come in the form of symbols representing functions. From symbols, tools either follow a sweeping or a recursive strategy [20]. In the sweeping strategy, tools first use symbols to mark an initial set of functions, then sweep the remaining bytes from the start of the file and mark sequences of bytes that resemble code as program code. In the recursive strategy, tools also start by using symbols to mark the initial set of functions, then locate other code sections following call edges and marking call targets as function entry points, hence program code. In some cases, uncharted regions in the binary are then plugged into machine learning algorithms to identify even more functions and code regions [19].

Functions are composed of one or more, usually several, *basic blocks*. A basic block is a sequence of instructions that contains no control flow instructions except as the last instruction of the block. If the first instruction in a basic block executes, it is guaranteed that all following instructions will execute. It is an abstraction that is used by many types of analyses. Once the functions are identified, their *Control Flow Graphs* (CFGs) are built. A sample CFG can be seen in Fig. 1. Building such a CFG correctly depends on correct identification of basic blocks and the edges between these basic blocks. Therefore, it requires the analysis tool to inspect each instruction in a function. This operation is error-prone, especially on variable-length instruction set architectures where an error in decoding an instruction propagates downstream and make decoding the following instructions harder, or even impossible.

3 Compiler Help

During the build process, compilers construct an internal representation of the source code and generate machine code using this internal representation. However, as soon as the executable file is generated, this internal representation is thrown away. In this paper, we investigate the effects of storing some of this information about the program inside the executable.

3.1 Basic Block and Edge Tables

To speed up building basic block abstractions inside binary analysis tools, we developed a compilation framework that stores the start address and the address of the last instruction of each basic block inside the executable in what we call a *Basic Block Table*.

CFGs require identification of edges between basic blocks. Our system stores the source basic block, the target basic block, and the edge type of

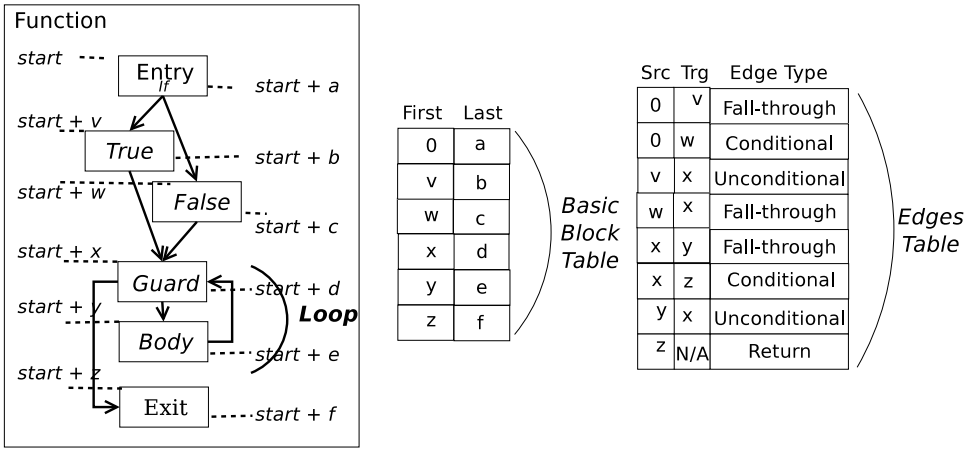

Fig. 1. A sample CFG and associated Basic Block and Edge Tables of a function with entry basic block, an If/Else structure, a loop, and an exit block.

each edge in the *Edge Table*. In this table, a basic block is identified by its start address. In some cases, target basic block in an edge can only be known during program execution. Such cases occur when the value of a function pointer or any other indirect branch target[1] depends on the user input. In these cases, we leave the target basic block field blank during generation of the Basic Block Table. Such edges can be filled in by the parser after the program has launched and when the value of the target address can be computed.

Figure 1 illustrates what information is stored in the Basic Block and Edge Tables based on the CFG shown on the left. For this example, assume *a, b, c, ...* are the offsets of the last instructions of their corresponding basic blocks from the start of the function. Since there are 6 basic blocks in the CFG, the Basic Block table has 6 rows. The first row represents the first basic block: it starts at offset 0, and its last instruction is located at offset *a*; the next row represents the second basic block, and so on. The Basic Block Table is followed by the Edge Table which has 8 rows, one for each edge between basic blocks. The edge from the entry block to the *else* block is represented with the triplet of *<0, w, Conditional>* since it is accessed by taking a conditional branch. The edge from the entry block to the *if* block is traversed when the conditional branch is not taken and when execution simply falls-through[2] to the *if* block. The last row in the figure represents the edge originating from the return instruction. Since the target of a return instruction cannot be determined statically, that field is left blank (marked with N/A in our example).

[1] Although indexed jump tables also use indirect branches, targets of such indirect branch instructions can usually be identified using heuristics.

[2] 'Fall-through' is a type of control transfer where the execution of an instruction is followed by the execution of the next instruction in executable's address space.

3.2 Compilation Process

Our compilation mechanism mimics a standard compilation process. We developed a tool that modifies assembly files and generates Basic Block and Edge Tables.

One issue in the implementation is the need to support position independent code such as libraries that can be loaded at different addresses in different executions of a program. Using absolute addressing does not work for this type of code. To handle this, the addresses in these tables are stored as offsets from the start of the function.

Another issue arises when some function definitions are merged by the linker. During the build process files that are linked with the *include* directive from a source file are compiled along with the actual source file. As a result all function definitions included from a header file are compiled into the resulting object file. If the same function definition is linked from multiple source files, these function definitions appear in multiple object files that result from the compilation of these source files. To avoid linking problems, these functions are marked as *weak*. Linkers allow only one copy of these weak functions to appear in the final executable - the remaining ones are dropped. Although these functions are identical at the source code level, since compilers perform optimizations individually on each copy of the function, the resulting machine code may be different. As a result, Basic Block and Edge Tables for these functions may differ slightly. Unlike weak functions, tables with the same name cannot be merged. Therefore, our compilation mechanism has to distinguish tables related to functions that have the same name. As a result, we generate table names using a combination of the function name and the name of the file that contains that function. At the end, each table has a different name, and there is a one-to-one mapping between functions and tables.

4 Evaluation

For evaluation, we used a simple binary modification tool we built based on Dyninst [4], an instrumentation and binary analysis tool for HPC applications. Our tool reads in a binary file and rewrites it to disk with simple instrumentation code for basic block counting. Since our analysis deals with every single basic block in the executable, the executable file has to be parsed from top to bottom, correctly locating every basic block. We compared the parsing speed of our tool with that of unmodified Dyninst. Although Dyninst is traditionally known for its dynamic analysis capabilities, it also serves as a static analysis tool with support for basic binary analysis and CFG generation along with binary rewriting. Like other static analysis tools, it makes use of symbols stored in the binaries to improve its analysis, although the existence of symbols is not required in many cases. Therefore, we expect our technique can improve similar static analysis tools in the same fashion.

We first evaluated our system on SPEC CINT2006 [9,15]. SPEC CINT2006 contains a series of CPU-intensive executables that are selected to evaluate the processor(s) and the memory system. All together, SPEC CINT2006 has about 1,047,000 lines of code.

Our next benchmarks were the PETSc libraries [2]. PETSc (Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It uses MPI for parallelization. It has linear and non-linear equation solvers and supports C, C++, Fortran and Python. The PETSc suite is composed of about 872,000 lines of code.

Finally we evaluated our system on the popular web browser Firefox (version 9.0.1) and all the shared libraries that ship with the Firefox source code. We evaluated our system on Firefox because its executables are numerous and are relatively large. Moreover, it contains hand-written assembly files and the build process involves using many uncommon compiler options. Therefore, building Firefox has been a valuable test for the robustness of our compilation mechanism. The Firefox suite contains approximately 5,335,000 lines of code.

4.1 Environment

All experiments were carried out on 64-bit x86 machines that run the Linux operating system. SPEC CINT2006 and PETSc benchmarks were tested on a system that has 4 Intel Xeon processors with 6-cores each and 48GB main memory. All our executables except PETSc were serial applications. Therefore, we ran most of our experiments serially on a single core. We used gcc 4.1.2 for building reference executables and as a back-end to our compilation mechanism. Firefox experiments were run on a separate machine due to the idiosyncratic requirements of the Firefox build environment. As a result, Firefox runs were taken on a dual-core machine with an AMD Turion processor at 1.8 GHz with 2GB main memory. On this system, we used gcc 4.6.1 for building reference executables and as our back-end. Since we never compare results across these machines directly, the results are not affected due to using two separate machines.

4.2 Experimental Results

Our first experiment was designed to calculate the time it takes to parse a specific executable using our analysis tool to show how much our tool improves parsing speed. We then ran other experiments to evaluate properties of executables built using our compilation mechanism and identify any trade-offs. Similarly, we compared file sizes after the compilation process. At the end, we tested the runtime performance of these executables in terms of time and memory usage. We ran each timed experiment 5 times and computed the mean. We then normalized our findings with respect to the executables compiled with the gnu compiler suite.

In our experiments, we observed that using basic block and edge tables reduced the parsing time between 58% and 77%, and on average by 73%. Although the file sizes increase by 23% on average, we believe this situation is not prohibitive since the basic block and edge tables are not loaded into memory during the execution of these binary files. We also observed about 23% increase in the compilation time. Since this is only a one time cost that appears while building executables, and it can be

improved drastically by integrating the creation of basic block and edge tables into the compiler rather than a separate assembly pass, we believe this increase is acceptable.

Experimental Parsing Results Figure 2 shows the normalized parsing times with SPEC CINT2006 benchmarks with respect to regular parsing. We observed a high percentage of speed-up across the board while the average binary parse speed-up is 3.7x (73% improvement over original parsing time).

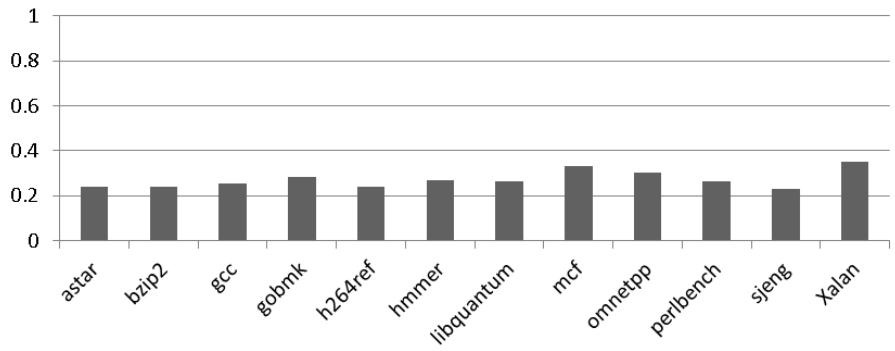

Fig. 2. SPEC CINT2006 Benchmarks: Normalized Parsing Times by Executable.

Our next tests were carried out on executables in *snes* package of PETSc suite. One interesting characteristic of PETSc executables is that PETSc libraries are statically linked into the executable files by default. As a result, each executable contains all functions in PETSc libraries. One can argue that the total cost of parsing these statically-linked executables is higher than the cost if these executables were linked with shared libraries. However, with the help of our compilation mechanism, we reduced the parsing time 76% on average (4.2x speed-up). Due to static linking of all these PETSc libraries in every executable, parsing time is more or less flat across all executables in this set because our tool parses mostly the same set of functions for each executable.

As a final set of executables, we decided to use the Firefox executable and all shared libraries that ship with Firefox. For this set of runs, we operated on those executables that reside in memory when the Firefox web browser is launched. We see a major improvement in parsing time once more as expected. The average drop in the parsing time is 71% (3.5x speed-up) with the worst case reduction of 58%.

As the previous results show, our system considerably increases the parsing speed. Now we want to discuss other evaluation metrics such as file size, compilation time, and memory footprint of executables.

Build Time Metrics Table 1 gives an overview of our experimental results regarding build time and runtime metrics. In this section, we will discuss the build time metrics: file size and compilation time.

Table 1. Various Properties of our System (All numbers are normalized)

Benchmark Set	File Size Growth		Compilation Time	Running Time	Memory Footprint
	vs. Standard	vs. Debug			
SPEC CINT2006	2.21	1.38	1.25	0.97	1.00
PETSc	1.50	1.09	1.32	0.95	1.00
Firefox	1.17	1.21	1.13	0.94	1.00
Overall Average	1.63	1.23	1.23	0.95	1.00

Since we are adding extra data to executable files, the size on disk unavoidably increases. On average, we are adding about 20 bytes of data to the executable for each basic block, and two extra symbols to the symbol table for each function. Table 1 shows the normalized file sizes across three sets of benchmarks along with the overall average. *Standard* shows the comparison of file sizes when they are built with no debug flag on while *Debug* shows the comparison with the debug flag (*-g*) on. We show both numbers since we realize executables are often built with debug flag on to improve debugging and other binary analyses on these files. The highest increase was observed by the SPEC CINT2006 benchmarks with 121% increase in the average file size with no debug flag on. On average, we observed an increase of 63% with no debug flag on, and 23% with the debug flag on. We assert that this increase is manageable since it does not impact the memory used during execution.

Another evaluation metric we used is the compilation time. Since our compilation mechanism uses an intermediate step to process the assembly code generated by the gcc, our compilations take more time than the original compilations. The bulk of the increase in compilation time comes from the cost of processing an assembly file as text, and writing out a modified assembly file, again as text. Currently, this step is costly and our experiments showed a 23% increase in the compilation time (Table 1). We believe our system can easily be integrated into a mainstream compiler such as gcc. Since compilers already maintain the information we generate using our mechanism, the added cost would be minimal - expected to be the same as the cost of writing those tables to the binary. This improvement remains as future work.

Running Time Metrics We next looked at the running times of the executables built using our compilation mechanism and their memory footprint. The results are presented in Table 1.

In our experiments we have not experienced any measurable increase in the execution time of the benchmarks. The slight improvement we observed in the running time after using our compilation mechanism is well within the noise of the experiment.

To measure memory footprint change, we evaluated each executable under Valgrind's *massif* tool [17] and measured heap memory and stack memory usage with *--pages-as-heap=yes* flag. Results indicate that both stack and heap memory used by the programs remain about the same, as shown in the final column in Table 1.

5 Related Work

Parsing binary code has been studied extensively in the past. Several researchers created higher level representations of machine code following binary parsing and the disassembly of code. Examples of this approach include Cifuentes and Gough with their decompiler, dcc [5], and Emmerik and Waddington with their Boomerang-based decompiler [7]. More recent work concentrates on disassembly of obfuscated code to identify malicious software [12]. Some researchers, such as Aaraj et al. [1], combine static disassembly techniques with dynamic analysis to cope with malware. Similarly, Bruschi et al. attempt to identify malware by building a CFG from binary code and comparing it with those of the known malware [3]. Disassembly techniques also made their way into the mainstream applications: Many common tools such as *gdb*, *objdump*, and *IDA* [10] generate disassembly of binary files. Many researchers build CFGs once the executable file is disassembled. De Sutter's [6] and Theiling's [23] control flow generation algorithms are such examples.

All these systems, including Dyninst [4], make use of the debugging symbols whenever possible. Many tools also perform a best-effort approach to identify function locations if the symbols are not present, such as Harris and Miller's tool [8].

With this work, we let binary analysis tools benefit from the knowledge compilers gather about executables during the build phase. There are several binary analysis tools that can make use of our system. Examples include ATOM [22], EEL [13], Pin [14], Valgrind [17], and Vulcan [21]. All these tools create some sort of internal representation of the binary. Therefore, they all can benefit from using the data stored in Basic Block and Edge tables as much as Dyninst does.

6 Discussion

Parsing executable files is the first step for any CFG-based binary analysis. Our experimental results show that our mechanism clearly speeds up parsing executable files. It is not hard to imagine bundling more information with the binary to speed up other binary analyses, or improve their precision, such as liveness analysis of registers or dependency analysis.

However, there are also shortcomings of our work. One such shortcoming is that our system adds $2n$ more symbols into the symbol table where n is the number of functions. Since symbol tables are highly-optimized, this issue is more of a nuisance than a technical problem. Increased compilation times might also be annoying for large frameworks such as Firefox.

However, we expect that integrating our system with a full compiler will substantially speed up compilation.

One improvement to our plain text table based system would be compressing Basic Block and Edge Tables to reduce disk space demand. In our preliminary experiments, we observed that compressing Basic Block and Edge Tables reduced the size of these tables by about 78%. However, since binary analysis tools cannot read compressed tables directly, they would need to decompress them before first use. We plan to investigate effects of using compressed tables in terms of parsing performance and disk space in our future work.

7 Conclusion

Parsing binary code is the first step for most binary analyses. However, it is costly and imprecise especially on variable-length instruction set architectures. In this work we introduced a novel compilation mechanism that improves the parsing speed of binary files when they are examined by binary analysis tools. Our compiler creates intermediate assembly files, augments them with information about basic blocks and edges between them, and generates executable files using this augmented assembly code.

We implemented an instrumentation program for basic block counting that rewrites a binary to the disk with the instrumentation code using the Dyninst library. We showed that running this analysis code on various benchmarks resulted in up to 4.4x speed-up in parsing time, with an average of 3.8x. Although the size of the binary files increase with extra data in the tables we generate, since these tables are not loaded into memory during execution, the size of the runtime memory image of the executable remains the same as before. Moreover, there is no runtime performance degradation due to these tables.

References

1. Aaraj, N., Raghunathan, A., Jha, N.: Dynamic binary instrumentation-based framework for malware defense. In Zamboni, D., ed.: Detection of Intrusions and Malware, and Vulnerability Assessment. Volume 5137 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 64–87
2. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2009) http://www.mcs.anl.gov/petsc.
3. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In Buschkes, R., Laskov, P., eds.: Detection of Intrusions and Malware & Vulnerability Assessment. Volume 4064 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 129–143
4. Buck, B., Hollingsworth, J.K.: An api for runtime code patching. Int. J. High Perform. Comput. Appl. **14** (November 2000) 317–329
5. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Software: Practice and Experience **25**(7) (1995) 811–829

6. De Sutter, B., De Bus, B., De Bosschere, K., Keyngnaert, P., Demoen, B.: On the static analysis of indirect control transfers in binaries. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA. (2000) 1013–1019
7. Emmerik, M., Waddington, T.: Using a decompiler for real-world source recovery. In: Reverse Engineering, 2004. Proceedings. 11th Working Conference on. (nov. 2004) 27 – 36
8. Harris, L.C., Miller, B.P.: Practical analysis of stripped binary code. SIGARCH Comput. Archit. News **33** (December 2005) 63–68
9. Henning, J.L.: Guest editor's introduction. SIGARCH Comput. Archit. News **35**(1) (March 2007) 63–64
10. IDA: About: http://www.hex-rays.com/products/ida/. Retrieved: March 21, 2012.
11. Ince, T., Hollingsworth, J.K.: Profile-driven selective program loading. In: Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I. EuroPar'10, Berlin, Heidelberg, Springer-Verlag (2010) 62–73
12. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of the 13th conference on USENIX Security Symposium - Volume 13. SSYM'04, Berkeley, CA, USA, USENIX Association (2004) 18–18
13. Larus, J.R., Schnarr, E.: EEL: machine-independent executable editing. In: PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, New York, NY, USA, ACM (1995) 291–300
14. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2005) 190–200
15. McGhan, H.: SPEC CPU2006 benchmark suite. Microprocessor Report (2006)
16. Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchithapadam, K., Newhall, T.: The Paradyn parallel performance measurement tool. Computer **28**(11) (nov 1995) 37 –46
17. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: In Third Workshop on Runtime Verification (RV'03). (2003)
18. Prasad, M., Chiueh, T.: A binary rewriting defense against stack based overflow attacks. In: Proceedings of the USENIX Annual Technical Conference. (2003) 211–224
19. Rosenblum, N., Zhu, X., Miller, B., Hunt, K.: Learning to analyze binary computer code. In: Proceedings of the 23rd national conference on Artificial intelligence - Volume 2. AAAI'08, AAAI Press (2008) 798–804
20. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02). WCRE '02, Washington, DC, USA, IEEE Computer Society (2002) 45–
21. Srivastava, A., Edwards, A., Vo, H.: Vulcan binary transformation in a distributed environment. Technical report, Microsoft Research (2001)
22. Srivastava, A., Eustace, A.: ATOM: a system for building customized program analysis tools. In: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, New York, NY, USA, ACM (1994) 196–205
23. Theiling, H.: Extracting safe and precise control flow from binaries. In: Proceedings of the Seventh International Conference on Real-Time Systems and Applications. RTCSA '00, Washington, DC, USA, IEEE Computer Society (2000) 23–