x86 function detection for Retrowrite
Semester project report

Sylvain Kuchen
EPFL
Lausanne, Switzerland
sylvain.kuchen@epfl.ch

Luca Di Bartolomeo (Supervisor)
EPFL
Lausanne, Switzerland
luca.dibartolomeo@epfl.ch

Mathias Payer (Supervisor)
EPFL
Lausanne, Switzerland
mathias.payer@epfl.ch

***Abstract*—Retrowrite[1] is a static binary rewriter for x64 and aarch64. Identifying which parts of a binary corresponds to functions in the source code is necessary for correct rewriting. However, Retrowrite currently only supports unstripped x64 binaries since it is unable to detect functions without symbols. We give Retrowrite the ability to work with stripped binaries with a new module that finds function starts. Our method combines strategies from recent works. We report a mean F1-score of 99.78% on a test set of more than 700 x64 C binaries.**

***Index Terms*—Retrowrite; binary rewriting; function detection**

I. Introduction

Function detection is a binary analysis technique that identifies which instructions correspond to original source code functions. Many binary analysis tool rely on accurate function detection for a variety of applications such as vulnerability detection [1], [2], security policy enforcement[3], [4] and binary rewriting[5]. Retrowrite belongs to this category of binary rewriting tools.

As of the time of writing, Retrowrite cannot rewrite x64 binaries that have been stripped, since it needs debug symbols to identify functions. Stripped binaries have had their non-essential symbols removed, usually to reduce their size. For this reason, binaries are most of the time stripped before being distributed. Adding the ability to detect functions in stripped binaries to Retrowrite allows the tool to operate on a much wider range of executables.

Our analysis procedure is based on Nucleus[6] and follows the same approach, but mixes in techniques from other works[7]. We do not rely on function signatures but on the CFG. The set of supported binaries is the same as Retrowrite's and is thus smaller: x64 ELF binaries resulting from the compilation of programs written in C.

A. Contribution

- We deliver a python module that identify function starts in x64 ELF binaries. While its original purpose is to be used in Retrowrite, it is sufficiently independent that it could be used as a library for other tools.
- We tested our system against a suite of 714 binaries and report a F1-Score of 99.8%. Moreover, 80% of the binaries in our suite are perfectly analyzed, meaning our tool does not produce any false positive or false negatives for them. We provide documentation and scripts to reconstruct the dataset and reproduce our results.
- We also lay out the work to continue the project to detect not only function starts but also function boundaries.

II. Background

A. Definitions

A **basic block** is a sequence of instructions which satisfies the 2 following properties:

- Instructions inside the block can only be executed sequentially.
- No other instruction executes between two instructions in the sequence.

The first instruction of a basic block is called its entry point and the last if called the exit point.

Using basic blocks are nodes in the **control flow graph (CFG)**. The edges between blocks can be classified based on the type of control flow transfer (call, jump, tail call,...) and its scope (inter-procedural or intra-procedural). The CFG opens up new possibilities to analyze control flow. **Reachability analysis** is done by depth-first search on the CFG by only following intra-procedural edges. It can be directed or undirected. When reachability analysis is started at a function start and is directed, it is also called **function body traversal**.

B. Challenges

Recovering function boundaries presents multiple challenges. We list some of them in this subsection.

Functions are generally entered by a call instruction, but can also be entered with a jump instruction. If the call happens at the end of the caller (tail position), the compiler will often perform *tail call optimization*. This saves a stack frame, since the callee reuses the caller's stack frame. This is problematic for function boundary detection: we cannot

[1]https://github.com/HexHive/retrowrite

rely on jump instructions being intra-procedural edges in the control flow graph. We need a way of identifying tail calls.

It is difficult to resolve targets of an indirect jump statically. There are existing techniques to recover jump tables and code pointers, but even recent work is not fully accurate[8]. Because of indirect jumps, it is not always possible to accurately construct the CFG.

Some functions are not reachable, or not directly reachable. They are indirectly called/tail-called. This makes them harder to discover.

Compilers all perform different optimizations, which can create unusual assembly or control flow. For example, `gcc` and `clang` can perform profile feedback-oriented optimizations[2], which identify parts of the code that are often executed (hot) and those which are seldom executed (cold). To improve locality, hot parts are compiled closed together and cold parts are placed away. They are linked together by direct jumps.

Some functions will trigger the program to exit and never return to the caller. Identifying such functions is necessary to produce a correct CFG. Moreover, the non-returning property can propagate: if a non-returning function is the only exit point of another function, then this function is also non-returning.

III. Method

Our analysis procedure is based on Nucleus[6] and follow the same approach. We do not rely on function signatures but on the CFG. We try to compensate weaknesses presented in the paper with other work[7] and some heuristics.

A. Disassembly

Disassembly is the process of recovering instructions from a binary. For this step, we rely entirely on Capstone[3]. We only disassemble the `.text` section. After this step, we get rid of nop instructions, since they create unwanted basic blocks.

B. Basic blocks and CFG

A simple procedure is used to group instructions into basic blocks. All entry and exit points are marked and instructions in between them are made into basic blocks. The edges between blocks are then resolved. This step does not take into account indirect jumps, and such blocks have no outgoing edges.

C. Function starts

From this point on, our analysis can be separated into multiple stages:

Direct calls: Call targets are definite function starts. Therefore, the first step is to iterate over all instructions and collect call targets.

Function pointers in other sections: `.init` and `.fini` sections both contain a function to be run before and after (resp.) the program is executed. The start address of these sections are also definite function starts. `.init_array` and `.fini_array` contain pointers to functions to be run before and after (resp.) the program is executed. These pointers are also definite functions starts.

Intel CET: Intel's Control-flow Enforcement Technology (CET) can be levereaged to find function starts. CET is a protection against return-oriented programming and call/jump-oriented programming. It introduces the `endbr64` instruction and ensures that indirect jumps and calls can only redirect to functions which start with this instruction. Thus, a very simple heuristic is to consider any `endbr64` instruction as the start of a function.

reg_tm_clones: Another useful heuristic concerns the detection of a particular function: `register_tm_clones`. This function is always tail-called at initialization. Although `register_tm_clones` is often found by tail call analysis, it is sometimes missed (in less than 0.02% of our tests). The call source is always `frame_dummy`, which is easy to detect using pointers in `.init_array`. We therefore simply detect `frame_dummy` and mark its call destination as a function start.

We will discuss the efficiency of these heuristics in sec. IV-D.

For the next stages, the more accurate the CFG the better, since they rely on reachability analysis. To this end, it is important to find and mark non-returning functions, as they introduce extra edges in the CFG if not taken care of.

To find non-returning functions. We start with a list of known non-returning function names, obtained from Ghidra[4]. Using the symbol tables of relocation sections, we find all non-returning addresses. We then iterate through all calls instructions and remove their fallthrough edge if their call target is one of those addresses. Through reachability analysis, we determine for each known function start whether they are still returning. If they are not, we add their address to our list. We repeat the procedure until no new non-returning function is found.

Tail calls: Tail calls are detected using reachability analysis and previously detected function starts. Reachability analysis is performed from every known functions starts. If there are gaps between reachable basic blocks, then jumps over those gaps are potential tail calls. Jumps before the

[2]https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization
[3]https://www.capstone-engine.org/

[4]https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Base/data/ElfFunctionsThatDoNotReturn

start address are also potential tail calls. However, we only classify them as definite tail calls if their target is a definite or potential function start. A potential function start is the instruction right after end of a contiguous sequence of reachable blocks starting at known function start. At the end of this stage, all tail calls destinations are added to known function starts.

This process is illustrated in figure 1. The area colored in yellow shows what instructions are reachable from `fun2`'s start. The darker area indicates the part where jumps are considered intra-procedural. Jump 1 is a potential tail call since it jumps before the function start reachability analysis was started at. However, the target is not a potential or definite function start. In contrast, jumps 2 and 3 both target potential or definite functions starts. They would be classified as tail calls. Jump 3 is considered a potential tail call since it jumps over a gap. It is possible that running reachability analysis again after tail calls were marked would yield more tail calls. For example, reachability analysis from the target of jump 2 might make the target of jump 1 a potential function start, and thus jump 1 a tail call. However, in practice, this usually produces more false positives because of profile feedback-oriented optimizations. This is because this kind of optimization creates many jumps similar to tail calls that end up being miss-classified. Our understanding is that there is no good way of differentiating these jumps from tail calls.

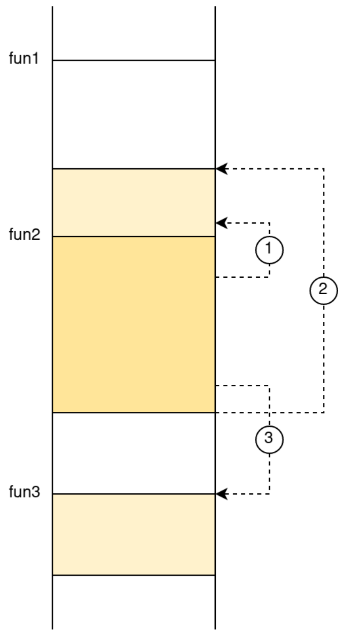

Figure 1: Tail call detection

Unreachable functions: The CFG is not always fully connected. Because we do not resolve jump tables, indirectly called functions will form separate components. We find those components and add function starts by examining the intra-procedural control flow. If a basic block has no incoming intra-procedural edges, we consider it to be the function start. When there are multiple such basic blocks, the one with the lowest address is chosen. If there are no such basic blocks, the basic block with the lowest starting address is chosen.

At this stage, our inability to resolve jump tables can create false positives: components being indirectly jumped to and non-returning will be considered unreachable functions. Switch statements often contains cases to exit in case of failure and thus create false positives. We use a simple heuristic to mitigate the issue, by checking if any of the previous basic block until the last known function start is ended by an indirect jump indirect. If such a block is found, we ignore the unreachable component.

This step is very similar to how unreachable functions are detected in Nucleus.

The combined function starts of all these stages is the result of the analysis. Note that it is possible for two stages to detect the same function start.

IV. Evaluation

A. Method

The function start detection module is implemented in around 600 lines of Python. It depends on Capstone for disassembly.

The system was evaluated using a dataset of 714 binaries downloaded from the Arch Linux `core` and `extra` debug repositories[5]. We chose Arch Linux debug repository since they organize binaries and debug symbols in a comprehensible way, in contrary to Debian for example. Since Retrowrite only supports C binaries, C++ binaries were excluded. This was done by checking if the section `.gcc_except_table` was present (indicating C++). Binaries in this repository are compiled with either `gcc` or `clang`.

Debug symbols were used to evaluate correctness. We extracted function names and start addresses from debug symbols. Function names ending with `.cold` were ignored. The reason is that the code they contain does not belong to a separate function in source code, but is the result of profile feedback-oriented compiler optimizations.

B. Summary of results

Disclaimer: The comparison between our solution and existing work cannot be fully accurate. Existing work does not use the same dataset for evaluation. [6], [7] use SPEC2006 (which contains C++ binaries) among others. Furthermore, other tools in our comparison are also applied to a broader set of compiled languages and multiple compiler optimization levels. For example, they were evaluated with C++ binaries, which are harder to

[5]https://europe.mirror.pkgbuild.com/

Function detection tool	Precision	Recall	F1-Score
Nucleus	0.9880	0.9780	0.9830
FIA	0.9960	0.9948	0.9954
Retrowrite	0.9991	0.9966	0.9978

Table I: Tool comparison (different datasets!)

correctly analyze. Our tool is designed for a more specialized usage, and is thus expected to perform well against more general solutions.

Results are shown in table I. Our implementation performs at least as well as other recent works. However, as disclaimed, we would need to evaluate the performance on the same dataset to have a fair comparison. We were not able to do this before the submission of this report. An important fact not present in table I is the fact that 80.1% of binaries in our test suite are analyzed without any false positives or false negatives.

C. Processing time

A more practical aspect is processing time. It takes our implementation up to 10 seconds for the bigger binaries in our test suite. A summary of processing by binary size is shown in figure 2. In comparison (disclaimer of previous section also applies), R. Qiao and R. Sekar[7] report a average processing time of 40 seconds per binary.

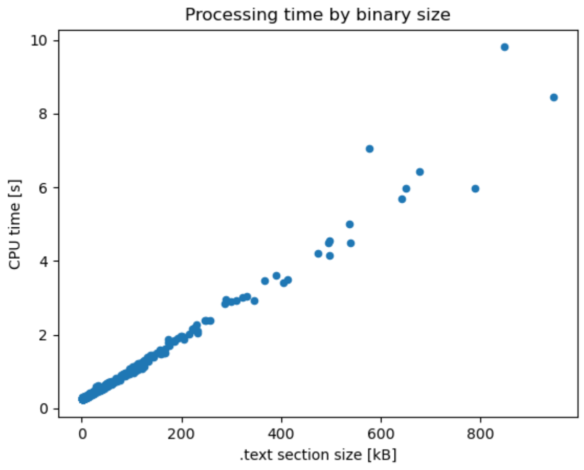

Figure 2: Processing time by binary size on Intel i7-8550U @ 4Ghz

D. Discussion of results

Definitions of metrics used in table II:

- Precision: $\frac{\#TP}{\#TP+\#FP}$, where $\#TP$ (true positives) is the number of function starts correctly identified by this stage and $\#FP$ (false positives) is the number of function starts wrongly identified by this stage. This metric is useful to understand how many false positives are produced by a particular stage.

Analysis stage	Accuracy	Coverage
Direct calls	99.99%	37.42%
Intel CET	99.91%	80.14%
.init / .fini	100.00%	3.77%
reg_tm_clones	100.00%	0.89%
Tail calls	99.81%	7.11%
Unreachable	33.88%	0.10%

Table II: Accuracy and coverage of each analysis stage

- Coverage: number of correctly identified functions by this stage out of the total number of functions to identify. This metric is useful to understand how many functions a stage can detect.

We discuss result details shown in table II:

- Intel CET `endbr64` are an excellent heuristic for function starts: more than 80% of all starts can be identified this way, with an accuracy of 99.91%. However, this might mean that our method will struggle with older binaries not compiled with CET.
- Surprisingly, there are binaries where targets of direct calls are not marked as function starts in debug symbols. We consider them edge cases.
- Unreachable function start stage has low accuracy. After investigation, we attribute the cause of this issue to missing edges in our CFG. In some cases, functions are erroneously marked as non-returning. This wrongly creates unreachable components. However, the distribution of false positives from this stage over binaries is very skewed in a few of them. Although it might be beneficial to disable this stage to improve the F1-Score, but in the case of Retrowrite accuracy is more important than recall.

False negatives are mostly due to working with an incomplete CFG, since we are not able to resolve jump tables. For example, tail call detection uses function body traversal to find non contiguous blocks. However, if the function contains indirect jumps, these bounds will be incorrect, and tail calls outside of bounds will not be examined.

V. Future work

There are a few possible ways to improve function start detection. Our approach does not resolve jump tables directly and instead deals with the possibility of the CFG being inaccurate. Adding the ability of resolving indirect jumps targets would lead to a more accurate CFG. In particular, this would decrease unreachable functions and non-returning functions false positives, and thus and overall reduced false positive rate.

Evaluation can certainly be improved: while our testing dataset contains a good number of binaries, we suspect our method might be over fitting our dataset. It is possible and probable that problems would be discovered if it had binaries from multiple sources. Ideally, research in this area could greatly benefits from an accessible, standard test suite of binaries of varying source languages, compilers and compiler options.

Of course, the next step would be to build on our current solution and move on to boundary detection. Since the current approach uses techniques that are applied to find boundaries in related work, we estimate that it is possible by extending rather than changing the approach. A challenge would be to define the notion of boundary more precisely and evaluate detection.

Another approach to detect function entries are call frames. Previous work[9] showed that this approach is promising. It would need more investigation, and might be combined with our current method.

VI. Conclusion

We hope that our work will contribute to making Retrowrite a capable and successful binary analysis tool.

Acknowledgments

I am grateful to Luca, who supervised the project. His advice has helped me focus on the right problems. I would also like to thank the author and contributors of the Capstone library. The quality of disassembly enabled me to focus on function detection.

References

[1] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, “Cross-architecture bug search in binary executables,” *it - Information Technology*, vol. 59, pp. 83–91, 2017.

[2] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A new approach to computer security via binary analysis,” in *ICISS*, 2008.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity[6],” in *Proceedings of the 12th ACM conference on computer and communications security*, 2005, pp. 340–353.

[4] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical context-sensitive CFI[7],” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 927–940.

[5] M. Prasad and T. Chiueh, “A binary rewriting defense against stack based buffer overflow attacks,” in *USENIX annual technical conference, general track*, 2003.

[6] D. Andriesse, A. Slowinska, and H. Bos, “Compiler-agnostic function detection in binaries[8],” in *2017 IEEE european symposium on security and privacy (EuroS&p)*, 2017, pp. 177–189.

[7] R. Qiao and R. Sekar, “Function interface analysis: A principled approach for function recognition in COTS binaries[9],” in *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, 2017, pp. 201–212.

[8] M. A. Ben Khadra, D. Stoffel, and W. Kunz, “Efficient binary-level coverage analysis[10],” in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 1153–1164.

[9] C. Pang, R. Yu, D. Xu, E. Koskinen, G. Portokalidis, and J. Xu, “Towards optimal use of exception handling information for function detection,” *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 338–349, 2021.