Analyzing the Performance of Block-Splitting in LLVM Fingerprinting

William Mahoney[1], Philip Sigillito[1], Jeff Smolinski[1], J. Todd McDonald[2] and George Grispos[1]
[1]University of Nebraska at Omaha, Nebraska, USA
[2]University of South Alabama, Mobile, USA
wmahoney@unomaha.edu
psigillito@unomaha.edu
jsmolinski@unomaha.edu
jtmcdonald@southalabama.edu
ggrispos@unomaha.edu

Abstract: This paper expands and builds upon previous work reported at the 2021 ICCWS concerning Executable Steganography and software intellectual property protection via fingerprinting. Software fingerprinting hides some type of unique identification into the binary program artifact so that a proof of ownership can be established if the artifact turns up elsewhere. In our previous work, it was noted that "fingerprints are a special case of watermarks, with the difference being that each fingerprint is unique to each copy of a program". This prior work emphasized making the fingerprint independent of the machine architecture; that is, performing the operations on an intermediate representation (IR). LLVM was used as the target IR, which is a compiler "middleware" language that is then converted into machine code in a later step. Both a static fingerprinting method, where the serial number or data is embedded and visible by inspection, and a dynamic method, where the code must be executed, were explored. The dynamic method only incurs an overhead if the proof code is executed and has very minimal impact if the proof code is not executed. However, the static fingerprint was accomplished by shuffling the order of basic blocks in the software in a manner that represents the serial number data, and this would have an impact on both the execution speed and the program size. This paper reports on subsequent research to improve the quantity of data which could be encoded by rearranging the blocks in a program and increasing the number of blocks by splitting them into smaller fragments, thus allowing for more potential orderings and therefore more data. Contributions in the current work are twofold. First, the experimental infrastructure has been refined so that the fingerprinting actions take place within the compiler itself as opposed to an external LLVM parser. Second, code has been introduced to limit the upper bound on the size of a block and to split blocks which are larger than this upper bound. We evaluate the resultant overhead and performance of the block splitting method and report negligible increases based on the block-splitting technique.

Keywords: Security and privacy, application fingerprinting, steganography

1. Introduction

Executable programs can be fingerprinted, making each copy of the software unique. This in turn provides for a certain amount of intellectual property protection, as a copy of the software appearing on eBay – for example – can be traced back to who the software was originally issued to. In fingerprinting, a unique serial number is embedded in some manner into the program. Thus, fingerprinting is a specialized use case for a digital watermark; a watermark can be used to prove authorship while a fingerprint can be used to prove ownership. A fingerprint *F* is embedded into a program *P*. There are two alternatives for this process: static fingerprints are detectable by an examination of *P*; it is not necessary to run the program to visualize the fingerprint. Contrasting with this is dynamic fingerprinting, where *F* is caused to appear only when executing a section of *P* with, for example, specific trigger data.

In a previous paper presented at ICCWS 2021 (Mahoney 2021), fingerprinting methods were described in detail. The paper described work to provide both static and dynamic fingerprinting methods. While the dynamic fingerprinting technique is not reconsidered in this paper, it was triggered by forcing a specific function to execute with a hidden flag set to a certain value. In this prior work the static method uses the serial number as a seed to a pseudo random number generator, which in turn dictates the order of the basic blocks within the program. Basic blocks are sections of executable code which are treated as a unit. Since the ordering of these units is directly impacted by the unique data used to start the random sequence, by examination of the binary artifact the ownership can be proven. All copies of a program will execute with the same expected outcome, but the ordering of the blocks of instructions within each copy is unique.

One aim of the previous work in fingerprinting was to add an additional aim to the set of goals normally encountered in this type of work. These original fingerprinting goals (Colberg and Thomborson 1999) are: 1) the fingerprint should be difficult to remove, 2) the fingerprint should be difficult to detect by the software owner, 3) the fingerprint should have a good data rate and a low false-positive rate, 4) the fingerprint does not severely

impact the performance of the program, and 5) the fingerprint is unique to each instance. The authors add an additional goal of making the fingerprint in such a way that it is language and machine independent. The LLVM intermediate representation in the compilation process is used as the point to interject the fingerprint.

Considering these goals, the ordering of the blocks cannot easily be altered by the program owner without significant investment in time and effort; the fingerprint is difficult to remove. Since the ordering of the blocks is dictated by a one-way function, going from this ordering back to the original serial number is difficult or impossible; the original serial number is not easy to detect. For goal number three, a simple way to think about the data rate is to consider the number of different messages which the program can surreptitiously contain. Of course, the number of possible different messages one can embed into a program by block shuffling depends on the number of blocks. For example, five blocks can carry 5!=120 different messages, while six blocks can carry 6!=720 messages. Large programs will be capable of a large quantity of hidden data (goal number three) and because of the growth in this number each program instance can be unique (goal five). This leaves the exploration of the performance impact.

As reported at the previous conference, the authors efforts at fingerprinting in LLVM were successful, but there was a slight impact on the performance of the program. This impact is contrary to goal number four, above. While not significant (4% was measured in the worst case) it begs a question. If one wishes to increase the number of blocks, to carry more messages, what is the resulting impact on the program performance, in terms of space and time? This paper addresses this block size/performance impact question.

The remainder of the paper is organized as follows: a brief description of the LLVM infrastructure is provide in the following section. Section three details other related work in this area, and section four describes previous work in this area by the authors of this paper. Section five contains the results obtained after the previous ICCWS paper. Conclusions and future work are in section six.

2. Background on LLVM

The original intent of the authors is to make the fingerprint in such a way that it is not machine specific. To do this we utilized the LLVM infrastructure. LLVM is the intermediate representation (IR) for several languages, most notably C and C++ but also including Swift, Rust, ActionScript, D, Fortran, Ruby, and many others (LLVM 2021). The LLVM project was started at the University of Illinois at Urbana–Champaign and originally stood for Low Level Virtual Machine (Adve and Lattner 2007). But this is now dropped and LLVM is considered *just* an acronym.

Languages supporting LLVM compile the program source code into the IR. Various tools exist for reading this intermediate code, optimizing it in some way, analyzing it from a machine code perspective, generating files which can be used to graph the control flow or dependence information, generating code analysis information, and other uses (LLVM 2021). Further, the IR can be translated from a bitcode format which can be interpreted, into a human-readable format, and from a human-readable format back to bitcode. The various compiled languages can generate the human-readable format and stop, making for easy inspection and learning about how the intermediate representation works.

Eventually, if needed, the IR is translated into machine code for a particular CPU, and the resulting executable program runs on that machine. This gives language and machine independence for fingerprinting, with the key being that one can compile the program to LLVM and then stop. The IR is then used to embed the data for the fingerprint, and the compilation process finishes by creating the executable program on the target machine. These CPUs supported by LLVM include ARM, x86-32, x86-64, IBM z/Architecture, and many others. LLVM IR can also be used to produce WebAssembly code, which can be interpreted by Microsoft Edge and Google Chrome web browsers. In the case the authors past work, LLVM code was successfully transferred compiled and fingerprinted back and forth between ARM7 and x86-64 CPUs, compiling to LLVM on one machine, fingerprinting, and then finishing on another machine.

3. Related Work

As mentioned above, fingerprinting is frequently either static or dynamic. Both these methods have been extensively studied by various researchers including Colberg (1999, 2002), Nagra (2002), and Palsberg (2000). Reordering of the basic blocks – like that used by the authors of this paper for static fingerprinting – was reported

on as far back as 1996 by Davidson and Myhrvold (Davidson and Myhrvold 1996). Other static fingerprinting methods include data hiding in the interference graphs of register usage (Qu and Potkonjak 1998) and embedding the information in a program's control flow structure (Venkatesan Vazirani and Sinha 2001). Stern has also used modifying the frequencies of the instructions to carry a message (Stern 1999).

In static watermarking, one interesting attempt was to hide executable programs inside of other executable programs by making the operands of certain instructions valid decodable and executable instructions, although the approach is quite limited (Mahoney et al 2018) (Mullins 2020).

In the realm of dynamic watermarking, an approach by Cousot assigns different values to variables as the program executes, and the combination of variable and their value represents the fingerprint (Cousot and Cousot 2004). The branching behavior of programs, the locations that they jump to, has been used to embed the fingerprint so that it is discovered via dynamic analysis (Colberg, Carter et al 2004). Wang, et al. (2018) encode the watermark by causing a program exception and monitoring the exception handling. Academics have also concentrated work in dynamic watermarking by making a software fingerprint distinct in terms of data structures (Kamela and Albluwib 2009) or using graph theory (Bento et al 2019) (Hamilton and Danicic 2020) (Colberg, Huntworth et al 2004).

A new method for dynamic fingerprinting is proposed by Alrehily and Thayananthan (2017) who use Return Oriented Programming (ROP). A consequence of using ROP is that the watermark is dynamic, and by the very nature of the ROP method the watermark is immune to anti-malware efforts such as address space randomization. Using opaque predicates, conditions that are known but must still be evaluated at runtime, Arboit (2002) has watermarked Java programs; this method is further studied by Myles (2006). A novel dynamic method is reported on by Ma et al (2019), using the Collatz conjecture. This conjecture is an unsolved mathematical problem which is used to change the control flow of the program in a detectable manner. Lastly, Preda and Pasqua (2017) explore the semantics of software watermarking as opposed to the methods.

The previous work by the authors of this paper appears to be one of few using the LLVM infrastructure; one other project with similar aims is the LLVM Obfuscator project (Junod et al 2015) which suggests similar benefits in terms of language and CPU independence, but with the focus on making the reverse engineering process more difficult. Their system includes obfuscation techniques such as substituting different instructions with the same semantics, removing cues that are present in the program control flow, and inserting erroneous (but never executed) instruction paths. There are no mentions of the performance impact, however.

4. Expanding Static Fingerprinting

In this follow-up research work the focus is strictly on the prior work with static – as opposed to dynamic – fingerprinting. Since the initial report the authors have modified the method used to introduce the fingerprint, so a review of the previous method and the new method is in order. In both cases the method utilized is based on previous work by Davidson and Myhrvold (1996). The fingerprint is created by modifying the ordering of the basic blocks within a program according to a set rule based upon the serial number of the software artifact. The serial number seeds a pseudo-random number generator which is used to dictate the order of the blocks. This method does not create a fingerprint that is necessarily reversible, but it does uniquely create a fingerprint based up on the executable's unique serial number. And the method does not impact the semantics of the program other than a potential degradation in performance, which is the point of this paper.

In the original implementation, a custom LLVM parser was created that could take a valid LLVM file and parse it into an in-memory parse tree. Programs that were to be fingerprinted were first compiled to LLVM, at which point the process is interrupted for the fingerprinting. The LLVM parser would then create a parse tree in memory and search the parse tree for functions. The types of the nodes within the data structure mimic the names in the grammar, so locating the body of functions is straightforward. The blocks within each function of the program can be determined, put into a collection, and then subjected to a Fisher–Yates shuffle (Fischer 1938) (Knuth 1969), and placed back into the IR tree representation. Once rearranged, a new LLVM intermediate file is created, and the compilation process is allowed to continue using the new IR file.

This initial method was working but turned out – over time – to be fraught with issues, making it less than ideal as a long-term solution. First, the grammar of LLVM seems to continually change from version to version, so

creating a viable grammar for the parsing of the LLVM proves to be like shooting at a moving target. Also problematic, LLVM uses a format called Static Single Assignment (SSA) which assigns unique names to all temporary locations used by the function and enforces the rule that no variable can be used before it is defined. Rearranging the order of the blocks keeps the same semantics but breaks the SSA rule. It is necessary to not only rearrange the blocks but traverse through the function, renaming all variables in a new ordering. But of course, different versions of LLVM have different rules as to what needs or does not need a variable number; every new version of LLVM would break the existing infrastructure.

For these reasons the approach was changed to modifying the actual compiler used – named Clang – to compile C and C++ programs into LLVM. The main advantage to this approach is that it almost singlehandedly eliminates the two problem areas in the prototype. First, if the language grammar for the human readable LLVM changes, one can use the parser integrated into the actual compiler which will of course match the latest changes in the grammar. Secondly, after the blocks are rearranged according to the fingerprint, there are already existing functions within the Clang code to renumber all variables in a function, making what used to be quite complex into a trivial function call. A command line parameter was added to Clang to turn on and off the additional various features to aid with testing.

Once this infrastructure was in place and working, the authors went back to refocus on the research question outlined previously. If one wishes to increase the data carrying capacity of a program, one way is to make blocks smaller. This results in a function having more blocks, therefore more potential permutations of the blocks, and thus a larger message carrying capacity. But this is at the potential cost in performance. Options were added to the modified Clang compiler to shuffle (or not shuffle) the blocks, and to "set the maximum size of a block to X". This latter rule causes the software to split blocks according to this maximum. A block with, for example, ten LLVM instructions but with a maximum block size set to six, will split the block into two blocks, one of six and one containing the remaining four LLVM instructions. A jump instruction added to the end of the first will transfer execution to the beginning of the second, regardless of what order the blocks might be in when converted to assembly.

5. Case Study

The original aim of the modifications was to split blocks into smaller fragments, to increase the data capacity of the program. The additional number of blocks directly impacts the data capacity by increasing the number of possible orderings of the blocks when they are shuffled. The larger number of blocks, when shuffled, should have a large impact on the performance of the program, both in terms of speed as well as size. To analyze these impacts requires two tools: a standard set of benchmarks and code to analyze the resulting executable programs made from these benchmarks.

5.1 Methodology

For measurements we turned to a subset of the "SPECspeed" benchmarks in the SPEC CPU 2017 benchmark package (SPEC 2017). The subset was selected according to two factors: first, the compiler modifications referred to above were in the Clang compiler and thus appropriate to C or C++ but not Fortran. Second, we want to have a good representation of the shuffling, and thus a program consisting primarily of (or almost exclusively of) external library functions were ignored. These restrictions leave eight benchmarks which seemed applicable for measurements, as well as an overall average measurement among this group.

It was also necessary in the prior study to create a program to analyze x86 instructions. This analysis program was written in Python, accepting the output of the standard Linux "objdump" program as its input. The program parses the objdump output and when it encounters a function, the algorithm checks that function signature against a blacklist. This blacklist contains regular expressions that match built in or global functions. If there is a match to the blacklist, the function is not considered in the totals, as these functions represent library elements which were not shuffled; C calls such as "printf" for example are exempt. If the function is not on the blacklist, then the statistics from the function are included in the data.

The benchmarks elected were compiled and tested on a HP ProLiant processor, 2.67GHz, with 160 GB installed, and a total of 24 cores available. Note however that while the multiple cores are used for faster compilation of the benchmarks the benchmarks themselves are not threaded and use only one CPU.

For technical reasons it is not possible to have a maximum LLVM block size setting that is less than three LLVM instructions. It thus seemed reasonable to test with the maximum block size ranging from three to ten, as via observation it did not appear that there were many LLVM blocks that were extremely large.

5.2 Performance Impact - Speed

To test the impact on execution speed, each of the benchmark programs was executed with a different maximum LLVM block size, ranging from three to ten. The execution runs were repeated 10 times and the number of seconds for each benchmark is the average of these times. (Note that benchmark 625 is three programs, and here the total execution time for all is shown. But 625 is broken out for the graphs concerning program size.) As expected, programs with more blocks execute more slowly, as the effects of the additional branch instructions and poorer cache performance due to a less optimum "principle of locality" have an impact. In figure 1 note the average speed (dashed line). The impact is not overly significant if the block size is limited to, say, six.

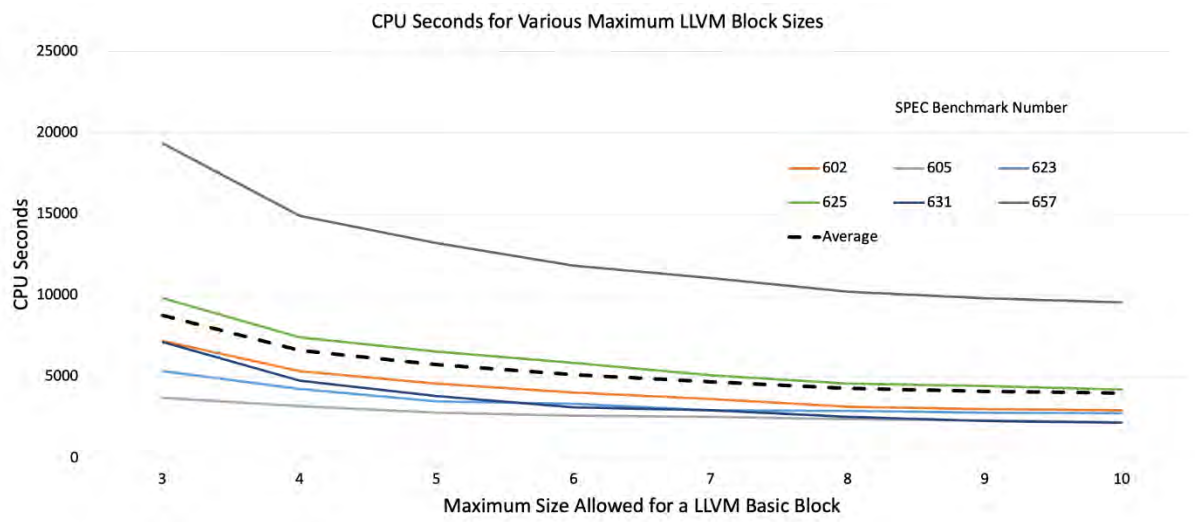

Figure 1: CPU Seconds Relative to Limited Maximum Block Size in LLVM Instructions

5.3 Performance Impact - Space

The authors next consider several effects in terms of program size. First, if one sets the maximum number of instructions in an LLVM block to N, the average number of instructions in a block will of course be less than N. Figure 2 below shows this impact. At lower values for N, three for example, the setting dominates because many or most blocks are larger than three instructions. But as the setting is increased, fewer and fewer blocks are as large as the setting, and the graph flattens. Eventually a larger and larger setting will have no impact once the setting is larger than any possible block in the program.

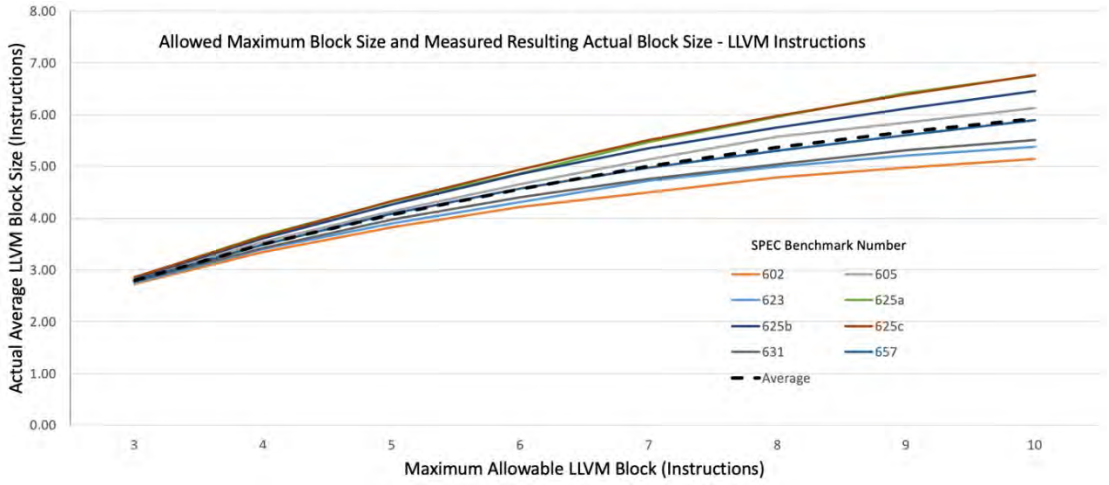

Figure 2: Average LLVM Instructions per Block as the Maximum Value is Increased

The next consideration is the relationship between LLVM block instruction count versus x86 block instruction counts. This is not a one-to-one relationship, because one LLVM instruction may produce more than one x86 instruction, or conversely two or more LLVM instructions may be converted into one complex x86 instruction. However, surprisingly the data indicate that after a point there is a close match between the average number of x86 instructions per block versus the original LLVM instructions per block. At a maximum block size of three there are 73% more x86 instructions than LLVM instructions – a high overhead. But at a maximum of 10 this overhead is down to just 5%.

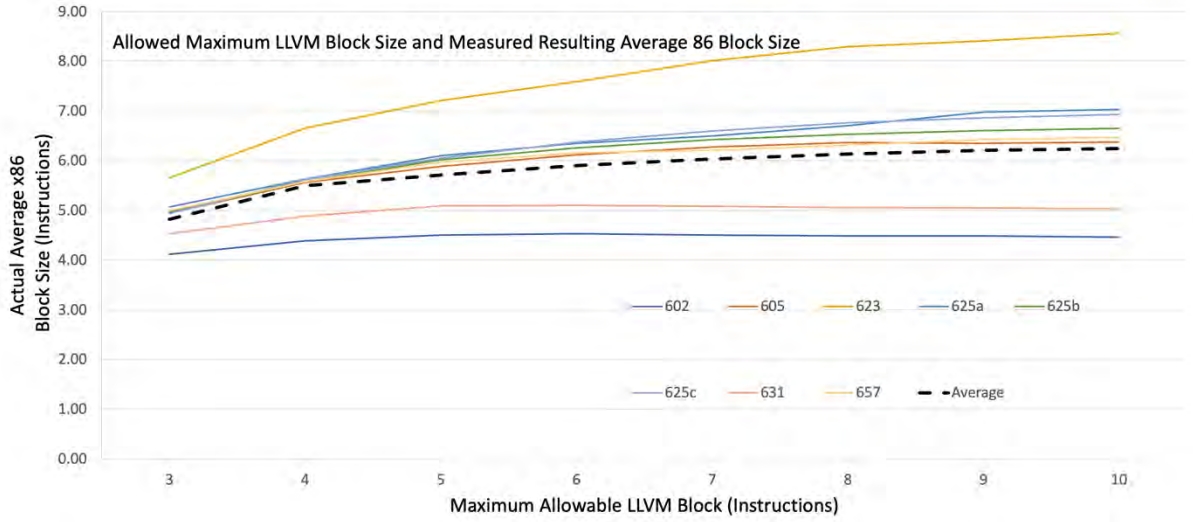

Figure 3: Allowed Maximum LLVM Block Size and Measured Resulting Average x86 Block Size

The final question is the number of actual machine code blocks that are generated based on different maximum LLVM settings. Again, this will not be a one-to-one mapping for several reasons. If after the blocks are shuffled, for example, two LLVM blocks end up in the proper sequence, the jump instruction from the first to the second block will be eliminated when the x86 code is generated. Recall that a motivation for smaller blocks is to increase the data carrying capacity of the fingerprint.

The number of x86 blocks generated from the compilation process, versus the maximum LLVM size, is depicted in the figure 4. The figure shows again that the maximum impact is with very small settings for the maximum number of instructions.

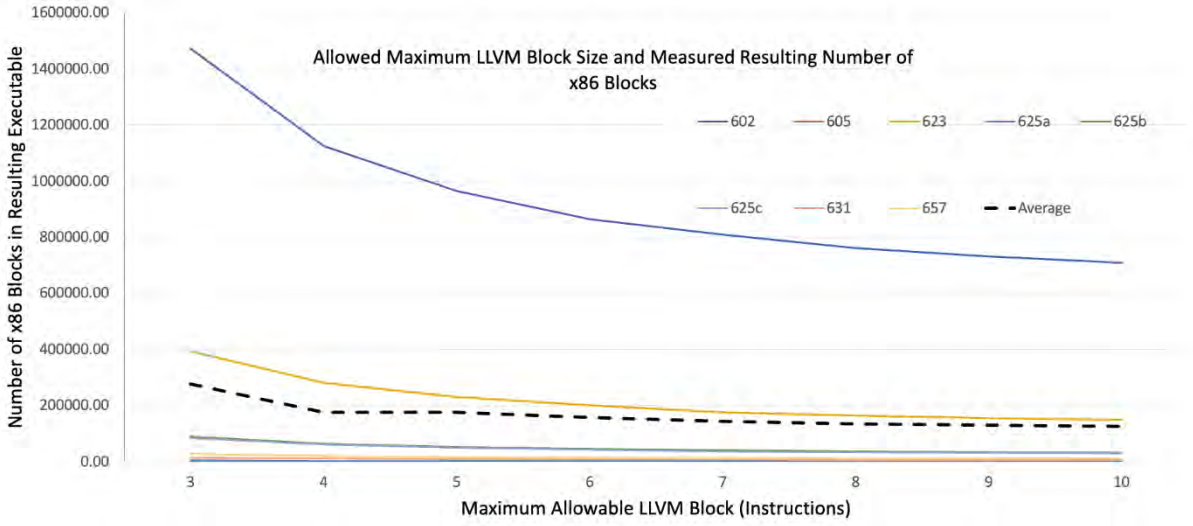

Figure 4: Allowed Maximum LLVM Block Size and Measured Resulting Number of x86 Blocks

As mentioned previously the data carrying capacity is dictated by the number of blocks. Recall that previously the example used was five blocks with a capacity of 5!=120, versus six blocks with a capacity of 6!=720 messages. The actual number of messages is less. Consider that in the process of making a permutation of blocks it is

possible that some of the blocks will be shuffled into their correct original ordering. When LLVM blocks are translated into x86 blocks, these adjacent blocks may be combined into one, removing the unnecessary jump instruction. One would expect the x86 block count to likely be smaller than the LLVM block count. Interestingly this is not the case, as shown in figure 5:

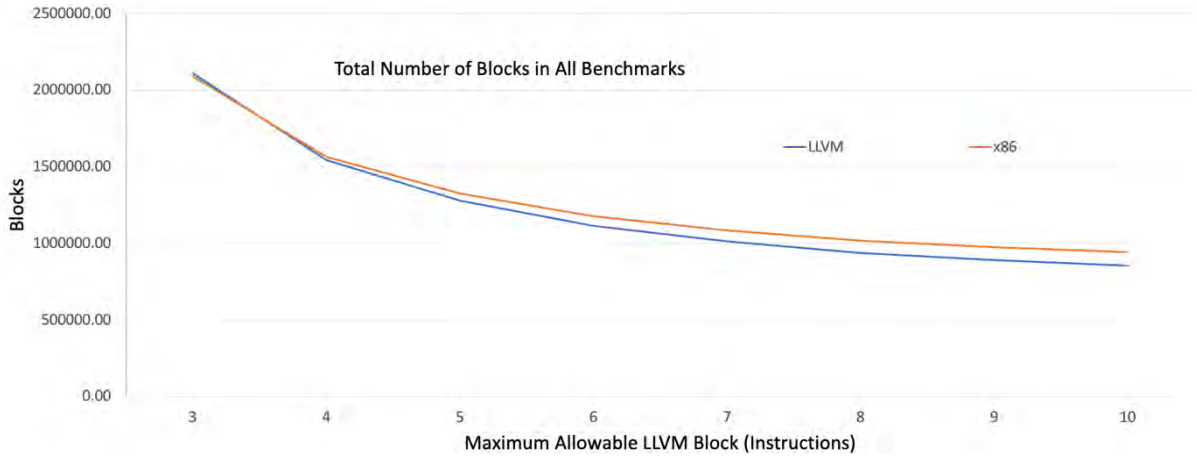

Figure 5: Total Number of Blocks in All Benchmarks – x86 and LLVM

We are investigating this discrepancy in more detail.

6. Conclusions

The use of block shuffling for software fingerprinting, using the LLVM infrastructure, has less of an impact in terms of size and speed as one might assume. However, there is a critical tradeoff that must be acknowledged. This tradeoff is related to the amount of information that can be "carried" (the number of permutations) versus the impact on the performance. This leads us to suggest that for fingerprinting purposes, one should select functions which are not frequently executed, but which have a higher degree of complexity. This greater complexity leads to a larger number of blocks, thus decreasing the need to split the blocks for fingerprinting purposes. Should the blocks need to be increased again, selecting an infrequently executed function will decrease any impact on the performance of the program.

In terms of execution speed, prior work (Mahoney 2021) indicated a speed overhead of approximately 1%. This was measured with no restriction on the block size. Our current work demonstrated a significant overhead when the maximum size of a block is severely restricted to a low number such as three or four instructions. In the case of limiting this block size to four, for example, the overhead is 66% (relative to a maximum of 10); but limiting the block size to six decreases this overhead to 29%. It thus would be possible to select a limiting factor based not only on the necessary data carrying capacity of a function (on the order of N! for a function with N blocks) but also with an eye to limiting the performance overhead at the same time.

In terms of space overhead, our prior work reported results that indicated a space overhead of approximately 1%, but with no limitation on the block sizes as we investigate in this paper. Decreasing the allowable size of an LLVM block will increase the space overhead of the resulting program as there will be a larger number of jump (branch) instructions in the resulting executable. An examination of figures 1 and 2 indicates that as the instructions per block limit is increased, the average instructions per block matches closely; this is true for the resulting x86 block sizes as well. Figure 3 shows that there is typically a larger actual x86 instruction count than LLVM instruction count. For example, if an LLVM block is limited to three, the resulting average in terms of x86 is around five instructions. However, as figure 3 indicates, this is only extreme at the lower values for this maximum. Figure 4 shows that the number of blocks (related to the data capacity, recall) does not change significantly if the limitation on block size is over four. Finally, a comparison of the number of LLVM blocks versus the number of x86 blocks (figure 5) shows that the two track rather closely. This combination of facts tends to say that restricting the block size within the LLVM intermediate representation will of course lead to a larger number of blocks and thus a higher data rate, but that the restriction likely should not be less than five or six instructions per block.

Lastly, the best method for increasing the data rate without suffering a significant overhead penalty might be to fingerprint only certain functions which are rarely called or only occasionally executed – startup code for the program, or exception handlers, for example, where the data rate can be increased by lowering the block size but with little performance impact. Given that the number of fingerprint combinations is directly related to the number of blocks and that this grows as a factorial function, it may be that this approach is the best for software intellectual property protection via fingerprinting with block shuffling.

Acknowledgements

This work is based upon work funded by the National Science Foundation under the Secure and Trusted Computing (SaTC) grants CNS-1811560 and 1811578. The project is a collaborative effort between the University of Nebraska at Omaha (UNO) and the University of South Alabama (USA).

References

Adve, V. and C. Lattner (2007) "2007 LLVM Developers' Meeting (Video)", [online], Available: https://www.youtube.com/watch?v=FNtmemyeEHY&feature=youtu.be.

Alrehily, A., and Thayananthan, V. (2017) "Software Watermarking based on Return-Oriented Programming for Computer Security", *International Journal of Computer Applications,* vol. 166, no. 8, pp. 21-28.

Arboit, G. (2002) "A Method for Watermarking Java Programs via Opaque Predicates", in *International Conference on Electronic Commerce Research (ICECR-5)* Montreal.

Bento, L. M. S., Boccardo, D. R., Machado, R. C. S., d. Sá, V. G. P., and Szwarcfiter, J. L. (2019) "Full Characterization of a Class of Graphs Tailored for Software Watermarking", *Algorithmica,* vol. 81, pp. 2899-2916.

Colberg, C. and Thomborson, C. (1999) "Software watermarking: models and dynamic embeddings", in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Antonio, Texas.

Colberg, C. S. and Thomborson, C. (2002) "Watermarking, tamper-proofing, and obfuscation - tools for software protection", *EEE Transactions on Software Engineering,* vol. 28, no. 8, pp. 735-746.

Colberg, C., Huntwork, A., Carter, E., and Townsend, G. M. (2004) "Graph Theoretic Software Watermarks: Implementation, Analysis, and Attacks", *Information and Software Technology ,* vol. 51, no. 1, pp. 192-207.

Cousot, P., and Cousot, R. (2004) "An abstract interpretation-based framework for software watermarking", *ACM SIGPLAN Notices,* pp. 173-185.

Davidson, R. I., and Myhrvold, N. (1996) "Method and system for generating and auditing a signature for a computer program". Patent 5,559,884.

Fischer, R. A., and Yates, F. (1938) Statistical tables for biological, agricultural and medical research (3rd ed.) London: Oliver & Boyd.

Hamilton, J., and Danicic, S. (2020) "A Survey Of Graph Based Software Watermarking", [online], Available: https://jameshamilton.eu/sites/default/files/GraphWatermarkingSurvey.pdf

Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J. (2015) "Obfuscator-LLVM -- Software Protection for the Masses", in *2015 IEEE/ACM 1st International Workshop on Software Protection*, Florence.

Kamela, I., and Albluwib, Q. (2009) "A robust software watermarking for copyright protection", *Computers and Security,* vol. 28, no. 6.

Knuth, D. (1969) Seminumerical algorithms. The Art of Computer Programming., Reading, MA: Addison-Wesley, pp. 139-140.

LLVM (2021) "opt - LLVM optimizer", [online], Available: http://llvm.org/docs/CommandGuide/opt.html and retrieved October 2021.

Ma, H., Jia, C., Li, S., Zheng W., and Wu, D. (2019) "Xmark: Dynamic Software Watermarking Using Collatz Conjecture", *IEEE Transactions on Information Forensics and Security,* vol. 14, no. 11, pp. 2859-2874.

Mahoney, W., Franco, J., Hoff, G., and McDonald, J. T. (2018) "Leave it to Weaver", in *8th Software Security, Protection, and Reverse Engineering Workshop*, San Juan, Puerto Rico.

Mahoney, W., Hoff, G., McDonald, J. T., Grispos G. (2021), "Software Fingerprinting in LLVM", 16th International Conference on Cyber Warfare and Security.

Mullins, J.A., McDonald, J. T., Mahoney, W., and Andel, T. (2020) "Evaluating Security of Executable Steganography for Digital Software Watermarking", in *Cybersecurity Symposium*, Moscow, Idaho.

Myles, G., and Colberg, C. S. (2006) "Software watermarking via opaque predicates: Implementation, analysis, and attacks", *Electronic Commerce Research,* vol. 6, no. 2, pp. 155-171.

Nagra, J., Thomborson C. D. and Colberg, C. S. (2002) "A Functional Taxonomy for Software Watermarking", in *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)* Melbourne, 2002.

Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao Q., and Zhang, Y. (2000) "Experience with software watermarking", in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)* New Orleans, LA.

Preda, M. D., and Pasqua, M. (2017) "Exception Handling-Based Dynamic Software Watermarking", *Electronic Notes in Theoretical Computer Science,* vol. 331, pp. 71-85.

Qu G., and Potkonjak, M. (1998) "Analysis of watermarking techniques for graph coloring problem", in *IEEE/ACM International Conference on Computer Aided Design*, San Joe.

SPEC (Standard Performance Evaluation Corporation) (2017) [online], Available: https://www.spec.org/cpu2017/

Stern, J. P., Hachez, G., Koeune, F., and Quisquater, J. J. (1999) "Robust Object Watermarking: Application to Code", in *Information Hiding; Lecture Notes in Computer Science volume 1768*, Berlin, Springer, pp. 368-378.

Venkatesan, R., Vazirani, V., and Sinha, S. (2001) "A graph theoretic approach to software watermarking", in *4th International Information Hiding Workshop*, Pittsburgh.

Wang, Y., Gong, D., Lu, B., Xiang, F., and Liu, F. (2018) "Exception Handling-Based Dynamic Software Watermarking", *IEEE Access,* vol. 6, pp. 8882-8889.