Software Fingerprinting in LLVM

William Mahoney[1], Gregory Hoff[1], J. Todd McDonald[2], George Grispos[1]
[1]University of Nebraska at Omaha, Omaha, Nebraska, USA
[2]University of South Alabama, Mobile, Alabama, USA
wmahoney@unomaha.edu
ghoff@unomaha.edu
jtmcdonald@southalabama.edu
ggrispos@unomaha.edu

Abstract: Executable steganography, the hiding of software machine code inside of a larger program, is a potential approach to introduce new software protection constructs such as watermarks or fingerprints. Software fingerprinting is, therefore, a process similar to steganography, hiding data within other data. The goal of fingerprinting is to hide a unique secret message, such as a serial number, into copies of an executable program in order to provide proof of ownership of that program. Fingerprints are a special case of watermarks, with the difference being that each fingerprint is unique to each copy of a program. Traditionally, researchers describe four aims that a software fingerprint should achieve. These include the fingerprint should be difficult to remove, it should not be obvious, it should have a low false positive rate, and it should have negligible impact on performance. In this research, we propose to extend these objectives and introduce a fifth aim: *that software fingerprints should be machine independent*. As a result, the same fingerprinting method can be used regardless of the architecture used to execute the program. Hence, this paper presents an approach towards the realization of machine-independent fingerprinting of executable programs. We make use of Low-Level Virtual Machine (LLVM) intermediate representation during the software compilation process to demonstrate both a simple static fingerprinting method as well as a dynamic method, which displays our aim of hardware independent fingerprinting. The research contribution includes a realization of the approach using the LLVM infrastructure and provides a proof of concept for both simple static and dynamic watermarks that are architecture neutral.

Keywords: Security and privacy, application fingerprinting, steganography

1. Introduction

Software fingerprinting is the process whereby a unique secret message (for example a serial number) is embedded stealthily within one or more copies of an executable program. The idea behind the the fingerprint is to provide proof of ownership, or lack thereof, for a specific binary artifact. Fingerprints are a special case of a digital watermark, where each fingerprint is unique to each copy of the artifact, allowing for this ownership tracking. Specifically, a fingerprint *F* is embedded into a program *P*; static fingerprints are those which are detectable or extracted by examination of *P* but without actually executing *P*, while dynamic watermarks are those that are caused to appear only by executing a part of *P*. In this paper we will use the terms watermark and fingerprint somewhat interchangeably, as the only significant difference for us is that a fingerprint is unique to each instance of *P*.

A good fingerprint is crafted in a way such that it is resilient and thus not easy to remove, well-hidden and thus not easily detected by the software owner, has a good data rate and thus a low false positive rate, does not impact the performance of the program and thus avoids notice, and is as unique as is necessary for each instance of the artifact (Colberg and Thomborson 1999). To this list we propose adding an additional desirable feature, that the process of fingerprinting software could or should be language and architecture-independent. To achieve this goal we propose the fingerprinting of software at the intermediate language stage of the compilation process using LLVM, an open source compiler middleware representation. Our desire is not to invent a new watermarking technique, as there are several researchers currently involved in this problem, but instead to investigate whether existing methods can be used to achieve language and machine independent fingerprints. Hence, this paper reports on our progress towards these efforts.

The remainder of the paper is organized as follows. In section two, we describe the open source compiler project LLVM, while section three details previous work in the domains of digital watermarking, software fingerprinting, as well as a previous research related to the use of LLVM to obfuscate executable programs. In section four we describe our approach, which uses block shuffling for static watermarking, and decryption of a hidden data string for dynamic watermarking. Section five presents the results of the evaluation of our approach including the

impact on the size and speed of programs. Finally, section six presents our conclusions and describes ideas for future research.

2. The LLVM Infrastructure

To implement our approach for an independent watermarking technique, we examined the LLVM infrastructure, currently used as the intermediate representation for compilers including the Clang C compiler, and others. The LLVM project started at the University of Illinois at Urbana–Champaign, initially as a research project to provide the middleware of a compiler system and was initially referred to as the Low-Level Virtual Machine (Adve and Lattner 2007). This acronym has subsequently been dropped and the now-non-acronym is used as a stand-alone moniker for the system.

The main component of LLVM is the Intermediate Representation (IR) which all of the language front-ends compile to and which is then optimized and used as the source for the various back-end CPU targets. The IR is strongly typed and features a RISC-like set of basic-like instructions that use Single Static Assignment (SSA) form. A key for us is the ability to stop the compilation process at the IR level, and this representation can be generated in a human-readable format. This output can be modified, and in fact can be converted back and forth to a bit format using a LLVM assembler or disassembler. When the program is in LLVM bit code it can also be executed using a just-in-time compiler or interpreter.

LLVM supports the compilation of several high-level programming languages into LLVM, including Ada, C, C++, Objective C, D, Delphi, Fortran, Haskell, Julia, Rust, and Swift. Various middle tools exist for taking this intermediate code and optimizing it, analyzing it from a machine code perspective, generating files which can be used to graph the control flow or dependency information, generating code coverage information, and so on (LLVM 2020). Since the IR representation is human readable, one can stop the compilation process, modify the readable version of the representation, and then continue the process to generate assembly code and eventually an executable. Numerous CPU architectures are available on this back-end, some with considerable support (ARM, x86-32, x86-64, etc.) and some with lesser or ended support (DEC Alpha). The IR can also be used to generate WebAssembly code which can be interpreted by web browsers including Google Chrome and Microsoft Edge. Thus, based on the above information LLVM was considered a strong candidate to evaluate our hypothesis that software fingerprinting can be made language and/or machine-independent.

3. Related Work

Software watermarking or fingerprinting is generally classified into two types: static watermarking and dynamic watermarking. An analysis of the literature reveals that both these methods have been extensively studied by various researchers including Nagra (2002), Colberg (1999, 2002), and Palsberg (2000).

Static fingerprinting schemes include hiding information within interference graphs of registers (Qu and Potkonjak 1998), hiding the information in the control flow structure of a program (Venkatesan Vazirani and Sinha 2001), and reordering the basic blocks of the program to encode the watermark (Davidson and Myhrvold 1996). An alternative approach proposed by Stern, et al. (1999) involves using the frequencies of the instructions in the program to encode the watermark with a spread spectrum approach.

Executable steganography, the hiding of software machine code in the operands of other software machine code, is a potential means to introduce new software protection constructs such as fingerprints. Initial work on the technique involved obscuring valid operation codes inside the operands of instructions (Mahoney et al 2018). However, while the method appears to work in theory, Mullins (2020) identified that the approach is difficult to implement largely due to the low likelihood of locating suitable instructions.

Researchershave also focused efforts on dynamic watermarking and making a software fingerprint more distinct in terms of data structures (Kamela and Albluwib 2009), or graph theory (Bento et al 2019) (Hamilton and Danicic 2020) (Colberg, Huntworth et al 2004). Cousot assigns distinct values to variables as the program executes, and the combination of variable and value is the watermark (Cousot and Cousot 2004). The branching behavior of programs has also been used to embed the watermark so that it is only known through dynamic analysis (Colberg, Carter et al 2004). Going further, Wang, et al. (2018) propose the use of branching, hereby encoding the watermark by exception handling (Wang et al 2018). Using opaque predicates, conditions that are known

but must still be evaluated at runtime, Arboit (2002) has watermarked Java programs and this method is further explored by Myles (2006). Alrehily and Thayananthan (2017) explored the use of Return-Oriented Programming (ROP) as an approach for software watermarking. A result of using ROP is that the watermark is dynamic, and by the very nature of the ROP method the watermark is immune to address space randomization. A new dynamic watermarking method is proposed by Ma et al (2019), using the Collatz conjecture, an unsolved mathematical problem, to alter the control flow of the program. Lastly, Preda and Pasqua (2017) explore the semantics of software watermarking.

In terms of watermarking or fingerprinting with LLVM, one project has a similarity to our proposed technique, but is used for different purposes. The LLVM Obfuscator project (Junod et al 2015) references the same advantages that we see in terms of font-end and back-end independence, but with a focus on the muddying of the executable code that is generated in assembly language. Their system includes control flow flattening, instruction substitution, bogus execution paths that are mixed with opaque predicates, and other methods aimed at making it more difficult to reverse engineer the binary executable.

4. Our Approach

In order to realize our approach, we aim to introduce both static and dynamic fingerprints into code compiled to the LLVM Intermediate Representation (IR). The summary of our approach is to create a LLVM IR parser, which can be used to read an IR file, build a tree representation of the intermediate format, conduct various transformations on this tree, and recreate a new valid IR file as the output. Hence, our proposed approach includes building the parser, creating the necessary tree transformations to insert static fingerprinting methods, and writing dynamic watermarking code which can be inserted into the IR so as to subsequently generate executable code.

4.1 Necessary Evils

An initial issue with creating the LLVM parser is the lack of a ‘standard’ in terms of grammar for LLVM IR files. The parsing within the LLVM system is not generated from a tool such as Bison (GNU 2020), which requires a high-level grammar definition, but by writing the syntax as C++ source code that implements the middleware. There are reverse-engineered IR grammars which can serve as a starting point (Eklind 2020) (Mewmew 2020), but these only support older versions of the LLVM project and lack some features that have been added in subsequent versions. It is thus necessary to hand-correct the grammar as newer features appear (and thus cause syntax errors). As an example, a certain C program when compiled with the ‘clang’ compiler creates a block in IR with the label ‘tag:’. However, the word ‘tag:’ is a reserved word in the grammar and thus ‘cannot appear’ as a label. These types of concerns are quietly ignored in the LLVM code but need to be handled if starting from an actual grammar specification.

As our research focus is somewhat of a feasibility study, creating the parser in order to conduct the experiments is a priority over writing (or adapting) something that is highly efficient. As a result, we elected to use PLY (Beazley 2020), a parser generator implemented in Python, taking the publicly available grammars as our starting point and fixing issues as they appear.

4.2 Static Fingerprinting

Our static fingerprinting method builds on previous research first proposed by Davidson and Myhrvold (1996). First, our code reads the IR for the program we wish to fingerprint. In the parse tree that results, we do not elect to remove any productions that result in empty terminals. In this way, the tree that results directly matches the grammar we started with at the beginning of the process. In particular, for any function definition in the IR, the function type, parameters, list of basic blocks, and other function specific information is ‘in the same place’ relative to the root of its tree.

Knowing this, we scan the IR in memory for the definition of each function and proceed to the blocks within the function. LLVM requires that the initial block of the function remain in place as it serves as the prologue for the function, setting up local variables. However, the ordering of the basic blocks within that function, not including the first block, are subjected to a Fisher–Yates shuffle (Fischer 1938) (Knuth 1969) and then placed in this order back into the IR tree representation. The Fisher–Yates algorithm relies on a sequence of random numbers, which are supplied by a standard Pseudo Random Number Generator (PRNG), and the seed for the PRNG serves as the

fingerprint for the executable code. That is, each issue of a software program can be made unique by seeding the PRNG with, for instance, the serial number of the software license.

This method does not create a fingerprint that is necessarily reversible. In other words, it does not answer the question of given an executable program, what is the serial number? But it does uniquely create a fingerprint based up on the executable's serial number. This process does not impact the semantics of the program other than a potential degradation in performance, which we will be discussed in subsequent sections of this paper.

4.3 Dynamic Fingerprinting

In the case of a dynamic fingerprint, we aspire to produce a fingerprint that provides a valid proof of ownership, but is not easily apparent through a static analysis. Our approach is to create a dynamic fingerprint function and compile it to IR, and then merge the fingerprint function into the software in such a way as to not impact the performance. A dynamic fingerprint is not normally executed. Hence, we expect the impact on performance to be minimal.

The first step in our approach is to describe a function of code that will represent the dynamic watermark. This function will generate a known output in a certain memory location if it is actually allowed to execute. The function is then merged into a similar function in the code we wish to protect. Since LLVM is strongly typed, a requirement is that the watermark function return the same data type as the merged function. It is anticipated that most programs, which are written in C or C++ would likely have functions with a return type of 'void', and a preliminary step in inserting the dynamic fingerprint is thus to locate 'void functions'. This is accomplished by parsing the IR and scanning the tree data structure, examining each function definition and printing the names of the appropriate functions. Then a function with these characteristics is arbitrarily selected as the destination for the dynamic watermark code.

Once a candidate function is chosen, the trees for both the IR of the program and the IR for the dynamic fingerprint are created, the target function is located, and the basic block lists for both structures are merged into one function. The next step is to add a preliminary basic block to select whether the fingerprint code is to be executed or whether the normal control flow of the function is to be executed. It may be necessary to insert a test against a known predicate, a volatile Boolean variable, in order to convince the LLVM conversion to not eliminate unreachable blocks. The final step is to shuffle the blocks so that the fingerprint code is intermingled with the original function.

A means is needed to display the fingerprint and thus prove the ownership of the code. Our fingerprint function is simply an implementation of the RC4 (Rivest 2014) stream encryption algorithm. Since RC4 is symmetric, encrypting a string into ciphertext and decrypting the ciphertext back into a string will produce identical results. Within the function we keep the cyphertext version of certain string data. As the program is executing, a debugger is used to trigger the watermark function. The resulting decrypted fingerprint can then be viewed in the program memory.

4.4 Experimental Methodology

For our concept we utilize several components, and a description of the flow of our experiments is in order.

First, as mentioned above, we used the PLY parser generator and implemented a parser for LLVM in Python using PLY. Different versions of LLVM have different syntax requirements, and also different versions of the 'clang' compiler generate different versions of LLVM. We decided to standardize with an up-to-date Clang and LLVM, but not a 'bleeding edge' system. Our Clang / LLVM version is 9.01. These were set up on Ubuntu Linux (version 20) machines. While a virtual machine was used for development use, a physical machine with no other users was used for the purpose of creating and measuring the benchmarks reported below.

For our first proof of concept, a simple open source web server named 'thttpd' (version 2.25b) (Acme 2020) was selected. This web server consist of approximately 12,000 lines of C code and includes header files that are easy to evaluate. The Makefile was modified to first compile the source to LLVM, then execute our fingerprint code on the LLVM.

In order to investigate the impact of our fingerprinting approach on program performance, we used the SPEC CPU 2017 benchmark package (SPEC 2020). While most of the of the 'SPECspeed' benchmarks are written in C or C++, one benchmark is written in Fortran and this benchmark was excluded from our analysis. The benchmarks rely heavily on configuration files, which provide the build process with information such as which compiler and linker to use in the process itself. Custom configuration files were developed that split and control the compilation process. This process includes compiling the source code to LLVM, executing the resulting fingerprinting test program, and then concluding the compilation into object files which are subsequently linked.

5. Results

While original objective is to determine the feasibility of static and dynamic watermarking in a language and architecture independent way, it is also important to investigate and acknowledge the impact on system performance as a result of the watermarking process. Hence, we evaluated our watermarking process using two approaches: by compiling and executing a simple web server with watermarking on multiple platforms, and by compiling standard benchmarks in two different programming languages.

5.1 Front/Back-End Independence

To evaluate language independence, as well as test the performance impacts of the technique, we use both the web server and benchmarks. As C and C++ are both well supported with the LLVM infrastructure, specifically the 'clang' compiler, running these tests validates our language-independence claim, at least for two high level languages. At the same time the nature of this benchmark package allows us to test the performance impacts in addition to demonstrating the machine independence.

We first address architecture independence, using two readily available platforms: a x86-64 personal computer and an ARM-based BeagleBoard (2018) computer. We first compiled our web server on the x86-64, generating LLVM, subjecting the IR to our block shuffling static watermark and then merged the dynamic function, and completed the building and testing on x86-64. Next, we moved the intermediate code from our x86-64 platform over to the BeagleBoard system where we completed the compilation with the architecture set to ARM and again successfully tested the server. We then moved the 'thttpd' source code onto the BeagleBoard system, and then built and tested the server with our watermarks. Finally, we brought the watermarked intermediate files from the BeagleBoard system back to the x86-64 and finished the compilation process . It must be noted that migrating the files back and forth between the x86-64 and BeagleBoard systems does not appear to impact the performance evaluation since the watermark is separate from the hardware platform.

5.2 Performance Impact

In terms of performance we first consider the resiliency of our watermark technique. The static watermarking consists of rearranging blocks in the physical locations within the application. Because the blocks are shuffled, we expect that there are fewer 'fall-through' blocks and instead anticipate that a larger number of blocks end in a branch or jump instruction. For the x86-64, rearranging the blocks (to disrupt the fingerprint) is difficult. First, the jump instructions themselves need to be modified since they encode a signed integer displacement in the instruction, which would no longer be correct. Second, the x86-64 contains different sizes of displacements. Rearranging the blocks may imply that a jump that contains a small displacement may now need a larger displacement, but there is no room in the instruction for this to be completed. For this reason, we anticipate that the static fingerprinting method on x86-64 is fairly resilient. In other words, it is difficult to rearrange the blocks once the executable is generated. Block shuffling on ARM is simpler than on x86-64 because of the fixed format of the instructions. In this case, because all the jumps are of the same size as the signed displacement jump, it would be considered much easier to calculate and replace the particular jump. Hence, shuffling on x86-64 is considered more resilient than shuffling on ARM-based systems.

As a test, we compiled the SPEC benchmarks with 'plain' options selected and then complied the benchmarks again with our shuffling of blocks. The aim was to determine whether there is a significant difference in the number of jump instructions in the resulting executable. A higher number of jumps would be an indicator that a program has been shuffled, as there would be fewer 'fall through' blocks in the program. Our results are in Table 1 - Jump Instruction Counts – Plain versus Watermarked presents the results of this analysis.

Table 1: Jump Instruction Counts – Plain versus Watermarked

Benchmark	Plain Option	Static Watermark Only	Static and Dynamic
600	79,092	79,793	79,856
602*	329,879	332,211	332,048
605	1,026	1,015	1,030
620	41,859	41,763	41,754
623	98,445	100,541	100,616
625	488 14,683 11,365	495 15,583 11,701	499 15,586 11,663
631	2,624	2,649	2,652
641	4,507	4,434	4,433
657	4,071	4,082	4,080
Total	588,039	594,267	594,217

*Test 602 is the GCC compiler, and there are two files in this suite that are sufficiently large which could not be processed by our tool.

The findings from Table 1 show that there was an approximate 1% to 1.5% increase in the number of jumps. Hence, these results suggest that unless the software owner can conduct a comparison between a plain compilation versus a static watermarked compilation, it will not be easy to detect the fingerprint.

In terms of our dynamic watermark, the fingerprint may be vulnerable. Consider that if one can locate the code for the RC4 decryption the following procedure could be used. First, replace the encrypted string built into the fingerprint with all bytes of zero characters. Next, allow or force the watermark code to execute and note what the decoded bytes contain. This will likely contain the bytes which the RC4 algorithm was going to exclusive-or with the cyphertext. Next, replace the cyphertext accordingly so that the fingerprint is changed. The key resiliency factor here is how easy it is to determine the location of the fingerprint code. The problem is that the end user has no idea what the fingerprint code does, and as such detecting where it is will be difficult.

In terms of data rate, the key performance metric is a low false-positive rate. In other words, what is the likelihood that a program fingerprinted with *F* embedded into program *P* ends up looking as if some other fingerprint F^1 was embedded into *P*? For shuffling, the factor to consider is the number of blocks in the program as a whole. To gauge this false positive rate, we examined our first program 'thttpd' to observe the number of blocks in the program which are eligible to be rearranged. The server contains 179 functions with a total of 3,087 basic blocks, thus 2908 blocks are eligible to be moved. This averages to 16 eligible blocks per function. An average function with 16 blocks would have 16! permutations, or around $2x10^{13}$. For even one or a few functions the chance of this 'hashing collision' is considered low, and it appears that the false positive rate could be zero. However, the resulting block permutation is actually linked to the seed value of the random number generator. We use the standard "srand" call in the Linux runtime library, which uses a 32-bit unsigned integer; thus 2^{32} or about $4x10^9$ permutations are all we should see in reality.

We next are concerned with the impact that the shuffling of basic blocks will have on the performance of the executable. More jump/branch instructions out to have some impact due to pipeline stalls, hardware branch prediction, and so on. To check this, we compiled the nine C/C++ SPEC benchmarks with the default options, again with blocks shuffled, and a third time with blocks shuffled and our dynamic watermark merged in. We expect the performance difference between these last two to be essentially zero; the dynamic watermark code will never be executed. Next, we ran each benchmark ten times in succession, and recorded the number of seconds needed for each of these groups of ten. Table 2 below reflects the results. We found the performance penalty to be fairly insignificant. In the worst case, SPEC benchmark 600, the overhead was 4%, and in fact for benchmarks like 605 the static watermarking code was slightly faster. Since this seems odd we repeated the test

several times and noted that even performing an identical test will have run times which vary by around 1%. This is a similar to prior work using GCC where the major impact to speed was not the block order but the use of indirect jumps to hide the control flow (Mahoney 2015).

Table 2: Seconds of Execution, 10 Iterations Per Benchmark

Benchmark	Plain Option	Static Watermark Only	Static and Dynamic
600	6271	6529	6559
602	8100	8160	8157
605	10479	10452	10288
620	6180	6181	6266
623	5020	5088	5080
625	4077	4019	4021
631	5120	5227	5153
641	6007	5976	5969
657	30433	30820	30800

We anticipate that the shuffling of blocks may have a minor impact on the size of the resulting executable file, and this proves to be correct. We ran the 'strip' command on each executable to assure that any symbols were removed from the files. One file in the 625 benchmark has a size overhead of 4%, but the remaining file size changes are not significant and in many cases the change is zero. We report these in Table 3.

Table 3: Executable File Sizes in Bytes

Benchmark	Plain Option	Static Watermark Only	Static and Dynamic
600	2439184	2455568	2463760
602	10722112	10767168	10763072
605	43408	43408	43408
620	2076800	2072704	2068608
623	4989048	5054584	5054584
625	31160 680480 663088	31160 717344 675376	31160 713248 675376
631	100784	100784	100784
641	177320	177320	177320
657	215632	215632	215632

An approach which might be used to determine the location of the dynamic watermark might be to compare code with and without the dynamic watermark function, in order to get an idea of the size of the opcodes making up the watermark code. However, selecting a few of the benchmark sizes, 600 or 625, for example, shows that the size difference is either zero or 4K, as the executable programs are made up of pages. The watermark code is sufficiently small that in some cases, benchmark 625 for example, the combination of rearranging the blocks plus adding the dynamic watermark actually decreases the size of the resulting program. Thus detecting the dynamic watermark by size difference is likely difficult.

6. Conclusions and Future Work

The use of LLVM as the intermediate representation does seem to be a logical choice to move forward with more advanced watermarking and fingerprinting algorithms. For example, more recent work in developing data structures as the dynamic watermark would improve upon the detectability, we noted in our selection of RC4.

Making use of the middleware approach definitely makes the static watermark portable across back-end machines, with the advantage of multiple front-end languages. However, we acknowledge this will only evaluated using the C and C++ programming languages.

Since this research is a proof-of-concept, the selection and implementation of PLY and a grammar for parsing and manipulating the LLVM middleware made sense for the purpose of this research. However, future work will evaluate the implementation of the actual LLVM source code parser with the aim of exactly matching the acceptable grammar characteristics of the intermediate representation, and improving overall speed. In addition, this enhancement could also improve the alignment of any occasionally floating requirements of the IR. This work is currently ongoing.

7. Acknowledgements

This research is supported 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

Acme (2020) "thttpd - tiny/turbo/throttling HTTP server", [online], Available: https://acme.com/software/thttpd/.

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.

BeagleBoard (1028) [online], Available: https://beagleboard.org/bone.

Beazley, D. (2020) "PLY (Python Lex-Yacc)", [online], Available: https://www.dabeaz.com/ply/

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.

Colberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn C., and Stepp, M. (2004) "Dynamic path-based software watermarking", in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, Washington D.C.

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.

Eklind, R. (2020) "A BNF grammar for LLVM IR assembly", [online], Available: http://lists.llvm.org/pipermail/llvm-dev/2018-June/123851.html.

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

GNU (2020) "GNU Bison", [online], Available: https://www.gnu.org/software/bison/

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 (2020) "opt - LLVM optimizer", [online], Available: http://llvm.org/docs/CommandGuide/opt.html

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. R. (2015) "Modifications to GCC for increased software privacy", *International Journal of Information and Computer Security,* vol. 7, no. 2-4.

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.

Mewmew (2020) "EBNF grammar of LLVM IR assembly", [online], Available: https://github.com/llir/grammar

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.

Rivest, R., and Schuldt, J. (2014) "Spritz—a spongy RC4-like stream cipher and hash function", [online], Available: http://people.csail.mit.edu/rivest/pubs/RS14.pdf

SPEC (Standard Performance Evaluation Corporation) (2020) [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.