OJIT: A Novel Obfuscation Approach Using Standard Just-In-Time Compiler Transformations

Muhammad Hataba, E-JUST
Ahmed El-Mahdy, E-JUST
Erven Rohou, INRIA

With the adoption of cloud computing, securing remote program execution becomes an important issue. Relying on standard data encryption is not enough, since code execution happens on remote servers, possibly allowing for eavesdropping from potential adversaries; thus the full execution process requires protection from such threats. In this paper, we introduce OJIT system as a novel approach for obfuscating programs, making it difficult for adversaries to reverse-engineer. The system exploits the JIT compilation technology to dynamically transform the code, making it constantly changing, thereby complicating the execution state. This paper quantitatively studies the effect of this approach by considering a set of obfuscation metrics borrowed from the software engineering field. The paper constructs a testbed system using the LLVM compilation framework that frequently applies random sequences of standard compiler optimizations on the currently running program. Results on using selected benchmarks from the SPEC CPU 2006 suite show a significant sustainable increase in obfuscation for a large number of standard optimizations over the runtime course of the programs.

1. INTRODUCTION

With the emergence of cloud computing, securing the execution of programs onto remote computers becomes an important issue [Zissis and Lekkas 2012]; an adversary can potentially inspect the execution state of the underlying virtual machine either directly (from the service provider side) or indirectly; via 'side channels' attacks (from the users side) [Ristenpart et al. 2009; Somorovsky et al. 2011].

There are many types of side-channel attacks: Trace-driven attacks aim to reconstruct the entire execution-trace of the program [Acıiçmez and Koç 2006; Kocher et al. 1999]; Access-driven attacks that analyze the correlation between processing and access patterns of physical resources such as the cache or memory [Neve and Seifert 2007; Gullasch et al. 2011]; timing-based attacks where an attacker monitors the execution time of a running program and utilize this to infer knowledge about operational dependences in the program [Brumley and Hakala 2009; Brumley and Tuveri 2011; Bonneau and Mironov 2006]; and more recently, acoustic attacks, which exploit the sounds emitted by computer internals such as the processor to recognize machine operations in an elaborate crypto analysis scheme to recover encryption keys [Genkin et al. 2013].

Author's addresses: M. Hataba and A. El-Mahdy, Computer Science and Engineering Department, Egypt-Japan University of Science and Technology (E-JUST), Alexandria-Egypt ; Erven Rohou, INRIA, Rennes, France.

[Ristenpart et al. 2009] have provided an interesting case study of such threats. They showed that a malicious insider in the cloud platform could harness the powers of the physical machine which he/she is co-resident in to launch side channel attacks. The attacker could model the cloud virtual machines (VMs) spatial information effectively, thereby exploiting placement vulnerabilities in cloud computing services providers. Thus adversaries can use this knowledge to plant their malicious VM with a victim instance, and then launch cross-VM side-channel attacks. [Ristenpart et al. 2009] launched their attacks against Amazon's EC2 cloud with success ratio of 40% in co-residence with a target VM.

These threats show that open system security principles – that is standard data encryption – is not enough; since the decryption software runs not the same platform, it permits eavesdropping from potential adversaries, with the aim of program reverse-engineering or even tampering. Thus "security by obscurity" is a promising paradigm, in which the designers hide system vulnerabilities in the system implementation. Hence comes the definition of code obfuscation as being the practice of hiding the purpose, meaning and operation of the code from adversaries, making it difficult to reverse engineer [Mataes and Montford 2005]. This concept is widely popular between virus and malware designers to hide their signatures from virus scanners to avoid detection and allow them to surreptitiously implant and execute their code on remote victims [You and Yim 2010]. Many real-world systems apply elements of these approaches to meet the aforementioned security requirements. This is related to what is called the fortification principle where "the defender covering all attack vectors" [Pavlovic 2011] via logical complexity [Aumann and Heifetz 2002].

In this paper we consider code obfuscation approach for securing such execution environment. Obfuscation allows for hiding the code semantics from adversaries; a plausible scenario is a company who wishes to adopt cloud computing so as to benefit from economics of scale, while making it difficult for adversaries to reverse engineer their programs. Here, we build on an earlier work [Hataba et al. 2013] to exploit the utility of just-in-time (JIT) compilers to dynamically and continuously change the execution image at rapid frequencies, thereby complicating the understanding of programs to the adversary. That is even if the attacker managed to launch a side-channel attack; they would have to collect all versions of the running program and try to find relations between these pieces, which we will conceal and try to make it as random as possible; thus rendering all analysis efforts useless. In particular, this paper has the following contributions:

— Extend an earlier LLVM implementation to allow for an automated testbed for full benchmarks (instead of selected kernels).
— Analyse the obfuscation strength of such approach on standard benchmarks from the SPEC CPU 2006 suite.
— Discuss the utility of such approach within the cloud computing environment.

The paper is organized as follows: Section 2 discusses some related work. Section 3 provides a background on LLVM compilation framework which we extend for obfuscation. Section 4 discusses the implementation of our system and the various obfuscation techniques we used. Section 5 introduces experiments and analysis of the system strength against different obfuscation evaluation metrics. . Finally, Section 6 concludes the paper and illuminates future work.

2. RELATED WORK

There have been many attempts in the literature to solve the security problem in the remote execution paradigm; however, none of them has considered using dynamic compilation technology.

Twin Clouds: [Bugiel et al. 2011] propose securing the cloud by utilizing two clouds as: a trusted private cloud (where the cloud is under user's control) and a commodity public cloud. The private cloud is used for encrypting critical data and algorithms (setup phase), and the commodity cloud is used for computing time critical computations (trusted cloud queries) in parallel under encryption (the query phase); a user first sends his/her request to the trusted private cloud which authenticates and encrypts the algorithm/data using a trust mechanism that is based on Yao's garbled circuits [Yao 1986]. Garbled circuits allow for computing a function between two parties without revealing a party's input to the other party. Garbled circuits symmetrically "encrypt" functions and securely evaluate it on "encrypted" data — we have a function represented by a garbled circuit and a garble table to map each garbled input to a corresponding garbled output. Then this can be applied on the encrypted data as well and, further more, can be verified. Each input '0' or '1' is assigned to a random (garbled) value. Then a table is constructed such that for each garbled input, a garbled output is calculated using a pseudo random function (garbled circuit). This process is based on two party encryption to realize what is called verifiable computing [Gennaro et al. 2010]. That is computing the value of a function with minimal knowledge from participating parties. The system exposes the twin cloud architecture to programmers, increasing the cost of software. Moreover, it incurs the extra cost of garbled circuit execution and communication between the clouds. But still this work presented a practically efficient approach for secure computations as opposed to Full Homomorphic Encryption (FHE) which aims to allow computations on encrypted data without using additional helper information [Gentry and Halevi 2011; Smart and Vercauteren 2010].

Hypervisor Security: [McCune et al. 2010] discussed the issue of how to trust a hypervisor. They present root trust static and dynamic management concepts. They suggest having a third party certificate authority that provides certificates that can be used for remote attestation of a given platform; by extending the Trusted Computing Base (TCB) as per the Orange Book [Latham 1986]. The difference between Static Root Trust Management (SRTM) and Dynamic Root Trust Management (DRTM) is that the latter can start a program in an Isolated Execution Environment (IEE) at any time not just at boot time, which is a new root for trust chained from the initial state of the machine (a clean CPU state). Hence, a client can be assured that its virtual machine is integral since it has started from a trustworthy state, and have not been modified or replaced by a malicious one.The system incurs a costly start overhead due to the chained trust mechanism. Moreover, the system is still susceptible to side-channels attacks from other virtual machines. Also a downside of the system is that the technique relies only on verifying that the hash belong to a list of trusted hashes, but that does not necessarily guarantee that it represents a trustworthy module. The certificate authority can be deceived by a fraudulent certificate issued by a malicious insider since the system relies only on a key for security in the launch process. Moreover, after the launch process, there is no way to guarantee neither the integrity nor the privacy of our computations on the cloud during runtime. There is also the risk of sabotage attacks via buffer and memory overflow exploits.

Secure Virtual Architecture (SVA): [Criswell et al. 2007] present a new compiler-based virtual instruction set for executing code on a given system including kernel and application code. The architecture provides instructions for object-level memory safety, control-flow integrity and type safety, allowing it to monitor all privileged operations and control physical resources. They also provide custom instructions to control memory layout such as allocation and explicit de-allocation instructions. Thus this work only protects the system from sabotage attacks on physical resource such as memory or buffer overflow attack. However, the system is still susceptible to eavesdropping attack, especially at the OS level. Moreover, SVA sandboxing mechanism focuses only

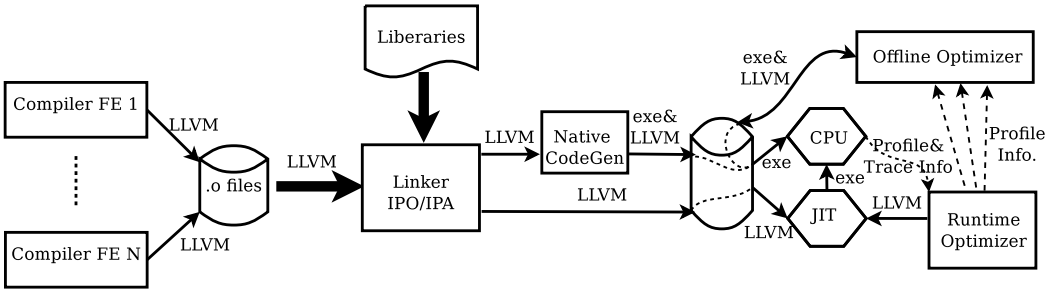

Fig. 1: LLVM System Architecture Diagram. [1]

on the instruction set beyond the Intermediate Representation (IR) level and a code-generation phase, which means – in our opinion – the system do not fully utilize the dynamic nature of LLVM's JIT compiler, which we consider in our system.

It is worth noting that obfuscation has been around for some time. It was introduced to manage the privacy of sensitive data in cloud computing platforms [Sharif et al. 2008] and for program protection as in [Hataba and El-Mahdy 2012]. But the more prevalent use of obfuscation is in hiding malware and other foist software in order to evade scanning or analysis. An example work of such context is of [Sharif et al. 2008].They proposed a technique for obfuscating trigger-based malware code, based on some conditions at the static compiler level. This scheme allows for evading malware analysis tools. They used LLVM compiler to transform the input program into an obfuscated binary. The system captures a conditional input trigger that starts the malware; it derives an encryption key from the input, encrypts the code, and then removes the key from the generated code. Thus analyzer programs cannot easily detect the start or execution of malware code. This system generates static obfuscated code essentially for malware triggering. However, our proposed system generates dynamically and every changing obfuscated code; which is generic in that sense.

3. OJIT IMPLEMENTATION

3.1. The JIT Compiler

A compiler is a special program that translates a program written in high-level language to machine-level instructions. The idea behind dynamic compilation is that a program is only translated to machine code at runtime as opposed to static compilers, which does the translation offline i.e before program execution begins. As for dynamic compilers we have two types: Ahead-Of-Time and Just-In-Time compilers. First type is Ahead-of-Time (AOT) compilation, in which compilation happens on the whole once during the start-up time, generating machine dependent binary. Another type is Just-In-Time (JIT) compilation [Aycock 2003] where program units (such as functions or basic blocks), are compiled on-demand, at the execution time. The generated code is machine dependent. Our OJIT system is based on JIT technology as it allows for continual recompilations, generating different binaries at runtime.

Program execution calls back (called trampoline code) into the JIT runtime system, that gives the JIT the advantage of adaptively optimizing programs taking into consideration the current execution characteristics such as cache hit ratio and instruction execution rate. That potentially allows for better optimizations than static compilers. Compilers have access to a vast amount of program semantic information. That, in addition to the dynamic nature of JIT compilers, has the potential for providing rich

[1] Original Figure from www.llvm.org, courtesy of Chris Lattner and Vikram Adve.

security objectives. We exploit the recompilation to constantly provide different programs, as will be described in the next section.

A quite powerful infrastructure for building compilers is the open source framework [The LLVM Compiler Infrastructure]. Historically, LLVM began as a research project known as (Low Level Virtual Machine) developed by Vikram Adve and Chris Lattner at the University of Illinois at Urbana, Champaign [The LLVM Compiler Infrastructure ; Lattner 2002] with the goal of providing a static/dynamic compiler applicable for an arbitrary wide range of programming languages. Now LLVM is the official compiler for Apple products including MAC OS X and iOS. The compiler is based on the famous Static Single Assignment (SSA) form, where a variable is assigned only once; SSA [Lattner 2002] significantly simplifies developing compiler optimizations.

3.2. System Design

Our goal is to design a system that produces functionally equivalent versions of the same program and have them running on the cloud in a randomly orchestrated yet ever-changing manner. That would make reverse engineering very complicated if not improbable. Hence, it will render side-channel attacks anything but useful.

Obfuscation can potentially be integrated into the original architecture of the LLVM compiler shown in Figure 1 along with its optimization stages as follows.

— **Front-End Obfuscation**
An Input program in any LLVM supported high level programming language such as C, C++, Objective C and Java can be subjected to various obfuscations on the layout level. These preprocessor transformations [Wroblewski 2002; Collberg et al. 1997] range from removing class information, scrambling identifiers, to inserting dummy code to confuse an attacker. Then the program is translated to bytecode (LLVM's Intermediate Representation (IR) code).

— **Back-End Obfuscation**
These could be during install time of the code to binary instruction corresponding to the target CPU architecture (currently supported platforms vary from typical x86, x64 architectures to smartphones' ARM processors and also Cell processors.). Possible obfuscations include changing address spaces, register reassignment or machine-level instruction set substitution.

— **IR LLVM Obfuscation**
This is our primary focus here; since we have the advantage of being largely machine and language independent. There are many obfuscation transformations that can be done to the IR code (see Appendix A which indexes a set of LLVM optimization passes that we used in our system and refer to [The LLVM Compiler Infrastructure] for more information on their operation and functionality). Some of these transformations can be used for the purpose of data flow obfuscation since change the data structure appearing in the source code: aggregating, eliminating or combining variables, propagating constants, sinking and re-association. Other transformation can be used to disrupt attacker by concealing control flow via obfuscation. Example techniques include: aggregation techniques such as: opaque predicates, loop unrolling, clone methods, inline/outline methods and changing the control flow graph (CFG). We selected a number of transformations that can be applied to mutate IR code dynamically. These transformations disrupt the otherwise normal behavior and control-flow of the program while retaining the same functionality. These transformations are randomly yet surreptitiously applied to the code during the JIT compilation on Function and Module Pass bases.

We modified the Execution Engine of LLVM forcing it to lazily call the JIT compiler every time a function is invoked. We combined this sort of trampoline calls with our own

obfuscation transformations, such that every time we JIT a function, we effectively obfuscate it.

There are many ways in which an attacker can analytically de-obfuscate our program in order to reverse engineer it [Kirk and Jenkins 2004]. Therefore, these techniques have to be taken into our design consideration as follows.

— Pattern Matching: Analyzers may compare and match obfuscated code to other well-known obfuscation techniques to find common similarities. So obfuscation has to be innovative and not at all generic, i.e. more syntactically related to the real program and randomly yet dynamically changing in that sense.
— Program Slicing: If we have used various methods of data and control aggregation and restructuring tools, a program may seem difficult to analyze by a human. But for a slicer program, it can carefully calculate variable values at every point of the program and which statements contributed to that value. In order to impede these tools, we can use inherent dependences between variables and add more bogus dependences thereby increasing slice sizes and making it more difficult for the analyzer to understand them. Also adding aliases for variable names or duplicate variables specially global and inter-procedural one; this will greatly slow down the slicer and slice sizes would grow exponentially.
— Statistical Analysis: De-obfuscators use statistical probability analysis to guess the outcome of predicates and evaluate conditional branches. So, predicates should be carefully selected not to be always true or always false. Several predicates can be chosen to have to be evaluated at the same time, for example, having side effects for opaque predicates.
— Data-flow Analysis: Usually de-obfuscation tools employ techniques that tend to eliminate obfuscations we introduced before. So we have to carefully choose what and where bogus code we inserted.
— Theorem Proving: For example, an analyzer can use a theorem proving mechanism to prove that some loop always terminates at a certain value. So we have to complicate that by employing more difficult to prove mathematical expressions.
— Partial Evaluation: Some tools can evaluate the static part of the program before runtime – sometimes pre-processing – and eliminate it. Introducing more inter-procedural dependences can impede this.

3.3. System Operation

The following are the normal steps upon which our system operates.

(1) Front-end compiler converts an input program into a LLVM intermediate representation object files (bytecode).
(2) Extract Symbols: dynamically link multi-phase communication between LLVM libraries and working module. Then create a stub for every symbol encountered. We here focus on function calls and create different stubs for each. These stubs will be evaluated later lazily.
(3) Identify machine code target for platform dependent optimizations and load the JIT execution engine.
(4) Lazily call the compiler: only when the execution requires that some function need to be computed, the stub is called and evaluated and also obfuscated. This makes sure that system does not fall into a circle of trampoline calls between the JIT and the executing program.
(5) Random number generation: choose an arbitrary random number that will be used to select the number of transformations suites to be applied and also which set of transformations will be selected. A strong random number generator forces unexpected code version. We can apply a series of K arbitrary transformation passes,

each of which can be chosen from the set of N obfuscation transformations (selected from the quite large available pool of LLVM optimisation passes);noticing that the order of applying these transformation is significant; therefore, we can generate $O(N^k)$ different transformation sequences. Thus, the probability of applying the same number and type of transformations in the same order is very rare to happen. An attacker would find great difficulty in enumerating all these possibilities, since theoretically K could be infinite.

(6) Select randomized transformations: we have a huge pool of selected transformations. Transformations order itself is random to add more obscurity.
(7) Symbol Resolution load required symbols "variable and objects" for the program and execute the program using the JIT emitter (backend compiler) which produces the machine code.
(8) Check for recursions: our scheme insures oblivious transformations even for recursive function call. Here we treat a recursive call as a new symbol and we create a new stub for it.
(9) Code morphing: this is a key feature in OJIT for producing multi-versions of the same code with the same functionality of course and remove old ones. As we said before we create a new stub for every symbol and even for recursive calls. After we use the code we remove its reference from the memory so that the memory space can be reused.

In order to make sure of the integrity of our obfuscation system and the correctness of the resulting code, we treat every function call as a new one and we obfuscate the source IR. After executing the obfuscated function, its pointer will be erased from the libLTO, the library class file that keeps track of all code-generated function. Thereby we force the compiler to treat it as a new function if it is called again anywhere in the program for example in a recursive call a situation typically happening in most programs due to what is called the principle of locality [Denning 2005]. The principle of locality states that the program stays 80% percent of the time executing 20% of the code.

As we said before the obfuscation unit for OJIT is a function call, it is also worth noting that there is a technique to extract loops as recursive function calls, which can be obfuscated afterwards dynamically using OJIT. Also we can extract a group of basic blocks into a new function call. Thereby we made the entire program as a series of function calls, which can cost a lot of branching overheads, memory exhaustion and then great performance degradation. Hence, for the sake of a simplicity and as a proof of concept we omitted these steps from the experimental evaluation.

4. EXPERIMENTS AND ANALYSIS

4.1. Choosing Candidate Metrics for System Evaluation

To evaluate the strength of obfuscation, we seek an objective evaluation method. Generally, assessing obfuscation is not trivial; there are four major criteria for such evaluation, namely: potency, resilience, software complexity and cost. However, the software complexity criteria is more suitable to provide for an objective measure. A suggested measure is to check for textual code dfferences [Collberg and Nagra 2009] which indadvertedly would translate to control-flow changes and data-flow dependencies thereafter. Much like the famous software plagiarism detection system [Moss (Measure Of Software Similarity)], we choose a sort of fingerprinting for 'similarity' [Schleimer et al. 2003] metric to measure the variations among the generated program versions. A well-known similarity metric is the longest common subsequence (LCS) [Chvatal and Sankoff 1975], widely used in the field of bioinformatics.

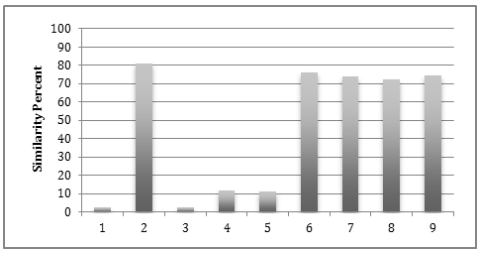

Fig. 2: LCSs for Compress Function in the bzip2 Program.

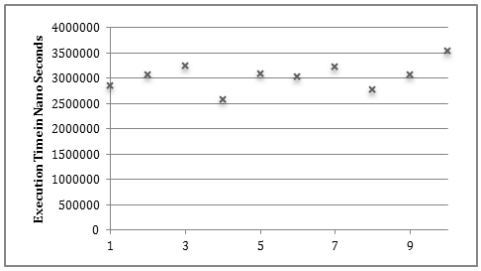

Fig. 3: Execution Times from Running the bzip2 Program 10 Times.

We also consider other software complexity metrics [Kirk and Jenkins 2004; Anckaert et al. 2007; Hataba and El-Mahdy 2012] which are: Instruction count (IC), branch count (BC) and the cyclomatic number, which is the number of merge points -as an indication of decision taken- (Phi). All of these metrics mainly assess the control-flow complexity. Also, we utilized execution time as an indication of cost, as well as obfuscation complexity.

4.2. The Experiments

We conduct our experiments on a MacBook Pro laptop, equipped with the Intel's Core i5 microprocessor, and 4GB RAM. The system runs Mac OS X Mavericks. We selected frequently executed functions from 401.bzip2, 470.lbm, and 473.astar programs (from the [Standard Performance Evaluation Corporation] SPEC CPU 2006 Benchmark Suite).

Figure 2 shows the LCS values for running the "compress" function of 401.bzip2 benchmark ten times. The achieved similarities varies from as low as 2% to as high as 80%. The choice of the set of transformations to apply is the main reason behind this variation. Figure 3 shows the variation in execution time; the results show around 30% variation with a cost reach up to a maximum of 15% as opposed to the original unmodified code version's execution time (shown in the first entry of the figure)

Since transformations significantly affects our obfuscation metrics, we consider the effect of each possible transformation in the LLVM system on a set of functions; the list of transformation is listed in Appendix-A. Since the effect depends also on the underlying program/function, we consider various functions. In the following, we ran the test program several times iteratively having it obfuscated using each transformation from the entire obfuscation pool of LLVM passes. Typically these benchmarks have a large number of function that are continuously obfuscated by our system. Therefore, as a proof of concept we focused on an arbitrary selected functions from these benchmarks and we collected our metrics from these specific observations.

For the 470.lbm program,as a show case, we focused arbitrarily on the two main functions: the entry *"main"* function, referred to in the figures as F1, and *"MAIN_parse CommandLine"*, referred to in the figures as F2. The results obtained are shown in Figures 4 through 7. Figure 4 shows the similarity (LCS) between a code version as a subject to an obfuscation transformation and the original code unchanged code version; similarities varies from 10% to 100%.

Figures 5 and 6 plot the results from the complexity metric IC, BC, and Phi counts; the variation is small for Phi counts, and significant for IC and BC with up to 40% variations, for both F1 and F2 functions.

The functions do not have loops, therefore they are not affected by loop transformation passes (27–30) (refer to Appendix A to learn the functionality of these transforma-

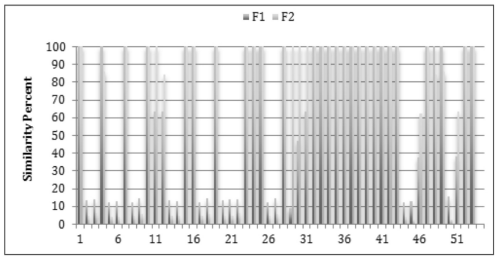

Fig. 4: LCS for the Main Two Functions in the lbm Program.

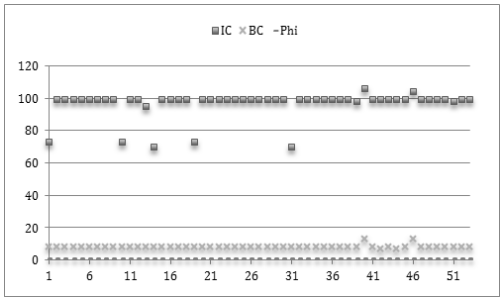

Fig. 5: IC,BC and Phi for the F1 Function in the lbm Program.

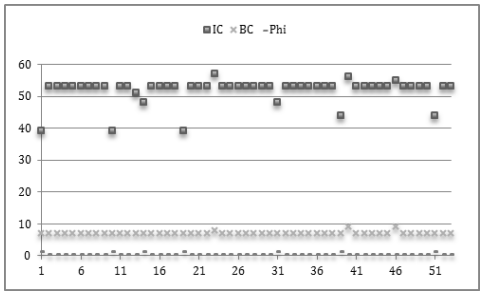

Fig. 6: IC, BC and Phi for the F2 Function in the lbm Program.

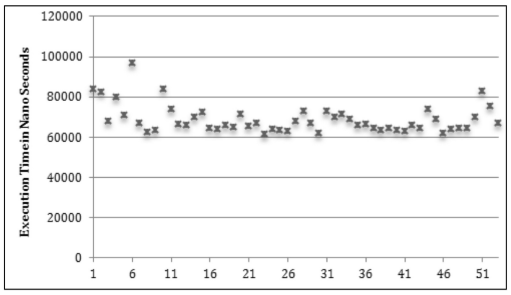

Fig. 7: Execution Times for the lbm Program.

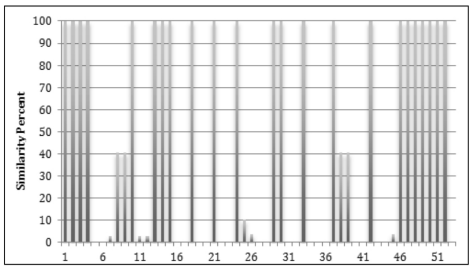

Fig. 8: LCS for the Main Function in the astar Program.

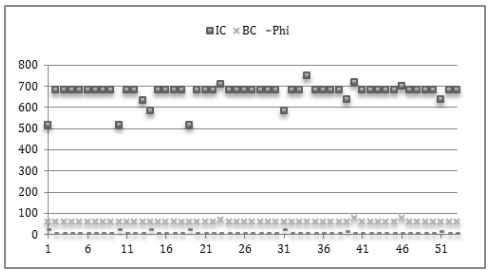

Fig. 9: IC, BC And Phi for the Main Function in the astar Program.

tions), see Figure 4. Also, for inter-procedural variable simplification transformations (2,4 and 5, - again see Appendix A- that eliminate some dead arguments or propagate constants), they have little effects on each function. That is mainly due to code simplicity, that does not involve much parameter passing. But other than such cases each program or even a function in it, will somehow respond to the aforementioned transformations. On the other hand, there are transformations that greatly affected each of the functions, especially the control-flow transformation passes such as: 5, 20, 21 and 22 (refer to Appendix A).These proved to be very useful and should be leveraged in an obfuscation-oriented set up.

Figure 7 shows total execution time for the 470.lbm benchmark after applying each transformation pass. Results show upto 40% variation. This gives a sense of cost to the

usage of each transformation pass. And comparing to the original program's execution time (shown in the first entry of the figure), the cost didn't exceed 20% . Nonetheless, such variation is a feature luckily helpful in impeding timing-based channel.

It is also worth noting that in the case of original OJIT such case would be improbable to happen. That is because each time we dynamically choose a random array of transformation passes to apply to the code at once and produce a corresponding code version. Such combinations of transformation passes would affect that, and thus provide resilience to various sorts of side-channel attacks and reverse engineering attempts.

Affirming the previous deductions, we put the 473.astar program into test with the same setup. This program has numerous functions, so we also arbitrarily focused on the entry *"main"* function. Figures 8 and 9 shows that the function have have rather complex control-flow, responding well to control-flow transformations.

It's worth mentioning that these obfuscation metrics -simple as they may seem- as well as others suggested in the literature [Hataba and El-Mahdy 2012; Collberg and Nagra 2009]are the subject of ongoing research to thoroughly assert our proposed system. In addition to that, a complete cost and performance measure of our system is being investigated. But for the time being the illustrated results are just preliminary observations to show the validity of our system. Also,undeniably the resilience of our system in real attack scenarios against automated analysis tools is an important evaluation criteria that we will investigate in our future work.

In summary, from the results it is shown that these transformations change the appearance of the code and hence its control-flow and data-flow dependences which translate to logical complexity and unintelligibility hindering attacker's reverse engineering goals via trace side-channels or decompilation tools. Also, we have to mention that execution time in our scenario incurs the initial overhead of JIT compilation as opposed to static compilation. Moreover, this overhead is combined with obfuscation transformations costs. Yet, these performance penalties, though might seem as a disadvantage, but in the context of timing side-channel attack scenarios these could be seen as a defence disrupting statistical analysis efforts. We foresee that the more dynamic and random are these changes being applied, the more secure our system should be.

5. CONCLUSIONS AND FUTURE WORK

This paper we explore the utility of JIT compilation to obfuscate programs. We have extend the LLVM compilation framework to allow for generating random sequences of compilation transformations. Using a set of obfuscation metrics and benchmark functions, the results of applying the method indicated significant increase in program obfuscation, reaching down to 2% code similarity, with a typical variation spanning 40% for our considered complexity metrics. The technique also has the advantage of being platform independent, and can readily be applied to remote execution environments, such as cloud computing.

OJIT is still in an early stage of development, and we plan to extend it along several dimensions. Future work include studying the effect of random transformation sequences in the front and back-end parts of LLVM. Moreover, explicit obfuscation methods can significantly further complicate programs, when combined with standard transformations, optimizations; this requires further investigations. Finally, an interesting aspect is modelling obfuscation cost (in terms of execution time) and strength trade-off, thereby allowing for controlling obfuscation depending on the required level of security.

REFERENCES

ACIIÇMEZ, O. AND KOÇ, Ç. K. 2006. Trace-driven cache attacks on aes (short paper). In *Information and Communications Security*. Springer, 112–121.

ANCKAERT, B., MADOU, M., DE SUTTER, B., DE BUS, B., DE BOSSCHERE, K., AND PRENEEL, B. 2007. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*. ACM, 15–20.

AUMANN, R. J. AND HEIFETZ, A. 2002. Incomplete information. *Handbook of game theory with economic applications 3*, 1665–1686.

AYCOCK, J. 2003. A brief history of just-in-time. *ACM Computing Surveys (CSUR) 35,* 2, 97–113.

BONNEAU, J. AND MIRONOV, I. 2006. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 201–215.

BRUMLEY, B. B. AND HAKALA, R. M. 2009. Cache-timing template attacks. In *Advances in Cryptology–ASIACRYPT 2009*. Springer, 667–684.

BRUMLEY, B. B. AND TUVERI, N. 2011. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*. Springer, 355–371.

BUGIEL, S., NÜRNBERGER, S., SADEGHI, A.-R., AND SCHNEIDER, T. 2011. Twin clouds: Secure cloud computing with low latency. In *Communications and Multimedia Security*. Springer, 32–44.

CHVATAL, V. AND SANKOFF, D. 1975. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 306–315.

COLLBERG, C. AND NAGRA, J. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection:*. Addison-Wesley, Reading, Massachusetts.

COLLBERG, C., THOMBORSON, C., AND LOW, D. 1997. A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand.

CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. 2007. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Operating Systems Review*. Vol. 41. ACM, 351–366.

DENNING, P. J. 2005. The locality principle. *Communications of the ACM 48,* 7, 19–24.

GENKIN, D., SHAMIR, A., AND TROMER, E. 2013. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Tech. rep., Cryptology ePrint Archive, Report 2013/857.

GENNARO, R., GENTRY, C., AND PARNO, B. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology–CRYPTO 2010*. Springer, 465–482.

GENTRY, C. AND HALEVI, S. 2011. Implementing gentrys fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011*. Springer, 129–148.

GULLASCH, D., BANGERTER, E., AND KRENN, S. 2011. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 490–505.

HATABA, M. AND EL-MAHDY, A. 2012. Cloud protection by obfuscation: Techniques and metrics. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*. IEEE, 369–372.

HATABA, M., EL-MAHDY, A., SHOUKRY, A., ROHOU, E., ET AL. 2013. Ojit: A novel secure remote execution technology by obfuscated just-in-time compilation. In *European LLVM Conf., Paris, France*.

KIRK, S. R. AND JENKINS, S. 2004. Information theory-based software metrics and obfuscation. *Journal of Systems and Software 72,* 2, 179–186.

KOCHER, P., JAFFE, J., AND JUN, B. 1999. Differential power analysis. In *Advances in Cryptology-CRYPTO99*. Springer, 388–397.

LATHAM, D. C. 1986. Department of defense trusted computer system evaluation criteria. *Department of Defense*.

LATTNER, C. A. 2002. Llvm: An infrastructure for multi-stage optimization. Ph.D. thesis, University of Illinois.

MATAES, M. AND MONTFORD, N. 2005. A box, darkly: Obfuscation, weird languages, and code aesthetics. In *Proceedings of the 6th Digital Arts and Culture Conference, IT University of Copenhagen*. 144–153.

MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. 2010. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 143–158.

MOSS (MEASURE OF SOFTWARE SIMILARITY). http://theory.stanford.edu/ aiken/moss/. Accessed Nov. 21, 2014.

NEVE, M. AND SEIFERT, J.-P. 2007. Advances on access-driven cache attacks on aes. In *Selected Areas in Cryptography*. Springer, 147–162.

PAVLOVIC, D. 2011. Gaming security by obscurity. In *Proceedings of the 2011 workshop on New security paradigms workshop*. ACM, 125–140.

RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 199–212.

SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 76–85.

SHARIF, M. I., LANZI, A., GIFFIN, J. T., AND LEE, W. 2008. Impeding malware analysis using conditional code obfuscation. In *NDSS*.

SMART, N. P. AND VERCAUTEREN, F. 2010. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*. Springer, 420–443.

SOMOROVSKY, J., HEIDERICH, M., JENSEN, M., SCHWENK, J., GRUSCHKA, N., AND IACONO, N. 2011. All your clouds are belong to us.

STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC Benchmarks, http://www.spec.org/. Accessed Nov. 21, 2014.

THE LLVM COMPILER INFRASTRUCTURE. http://www.llvm.org/. Accessed Nov. 21, 2014.

WROBLEWSKI, G. 2002. General method of program code obfuscation (draft). Ph.D. thesis, Citeseer.

YAO, A. C.-C. 1986. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 162–167.

YOU, I. AND YIM, K. 2010. Malware obfuscation techniques: A brief survey. In *BWCCA*. 297–300.

ZISSIS, D. AND LEKKAS, D. 2012. Addressing cloud computing security issues. *Future Generation Computer Systems 28,* 3, 583–592.

Appendix A

The LLVM uses the list of transformation as shown in the following table. We label each transformation with a number, that is referenced earlier in the paper.

Transformation No.	Optimization Pass Name
1	Reassociate Pass
2	GVN Pass
3	CFG Simplification Pass
4	Constant Propagation Pass
5	Instruction Combining Pass
6	Dead Instruction Elimination Pass
7	Dead Code Elimination Pass
8	Aggressive Dead Code Elimination Pass
9	Demote Register To Memory Pass
10	Promote Memory To Register Pass
11	Memory Copy Optimization Pass
12	Dead Store Elimination Pass
13	Indirect Variable Simplify Pass
14	LICM Pass
15	Sinking Pass
16	Instruction Namer Pass
17	AAEval Pass
18	De-linearization Pass
19	Partially Inline Lib. Calls Pass
20	Unreachable Block Elimination Pass
21	Early CSE Pass
22	Lower Expect Intrinsic Pass
23	Instruction Simplifier Pass
24	Jump Threading Pass

25	Scalar Replicate Aggregates Pass
26	Tail Call Elimination Pass
27	Loop Rotate Pass
28	Loop Unswitch Pass
29	Loop Deletion Pass
30	Loop Unroll Pass
31	SCCP Pass
32	GC Lowering Pass
33	Dwarf EH Pass
34	Stack Protector Pass
35	Flatten CFG Pass
36	Break Critical Edges Pass
37	Default PBQP Register Allocator Pass
38	Basic Register Allocator Pass
39	Greedy Register Allocator Pass
40	Fast Register Allocator Pass
41	Lower Switch Pass
42	Lower Invoke Pass
43	Function Inlining Pass
44	Sample Profile Loader Pass
45	IPSCCP Pass
46	Lint Pass
47	Argument Promotion Pass
48	Atomic Expand Load Linked Pass
49	Constant Hoisting Pass
50	Global Optimizer Pass
51	Scalarizer Pass
52	Add Discriminators Pass
53	Separate Const. Offset From GEP Pass
54	Lower Aggr. Copies Pass
55	Allocation Hoisting
56	Global Base Register Pass
57	Cleanup Local Dynamic TLS Pass

Table I: List of used transformations