The University of Auckland

COMPSCI 780

Benchmarking software obfuscators
An experimental approach

Rafik Nasraoui

supervised by
Dr. Clark Thomborson

May 11, 2016

Abstract

The optimisation of software code execution speed is a key consideration in any software project. A high level of security, to the extent that all hacks are rendered impossible, is also of crucial importance to most software. One way of protecting sensitive information is through obfuscation, which aims to make the computer programme unintelligible to a human being, without altering the core functionality of the programme. A problem with obfuscation, is that it may slow down the processing speed of a software, and therefore reduce the overall user experience. We analyse the results of a preliminary experiment that benchmarks the execution time of a fixed-width boolean circuit, where the width of the circuit is increased with each execution. The circuit represents an approximation of a result of a functional encryption operator, i.e. of an obfuscation. The experiment collects cache data generated by the executing systems hardware performance counters, to measure the execution time of the circuit.

1 Introduction

Security and data privacy are critical aspects of most software programmes. The need for such security can derive from the need for compliance with government regulations, and individual and company privacy requirements. The protection of sensitive information and industry data is also vital for software to stay competitive. The ability of hackers to extract sensitive data is continuously advancing, and so software data protection must also continuously evolve, in order to keep ahead of such efforts.

One of the mechanisms that could be implemented in the future to protect data is obfuscation. Software obfuscation aims at making computer programs unintelligible for a human being, without affecting the functionality of the software. Although software must be delivered into the hands of the public in order for the public to make use of the software, obfuscation hides the information contained within the software, including the code itself, by rendering it illegible to a user or hacker.

Obfuscation has potential useful applications in preventing reverse engineering and thereby protecting Intellectual Property, and in enabling secure software distribution. Obfuscation can also be a powerful means of encrypting data.

Until recently, there was no formal definition of obfuscation, and all techniques, including manually obscuring data, could be given the label obfuscation. Recent developments in cryptography theory have however produced a formal definition of obfuscation, which will most assuredly lead to an increased interest in the field, in the hope that obfuscation can be practically applied to protect software in the near future.

However, one of the drawbacks of obfuscation is that it may be ultimately limited by the hardware on which the software is run. Obfuscators

need to produce obfuscated programs using available computing resources in a commercially relevant amount of time. These obfuscations then need to be evaluated by end users in relatively similar, if not more challenging conditions, in terms of hardware performance and functionality. Normal encryption methods in general have small variable size and a low number of instructions, and therefore require little run time to execute. Obfuscators however are technically challenging to implement, and the number of variables and instructions arising from an obfuscation may surpass the limits of the hardware within which the software is run.

This paper aims to test the hypothesis that obfuscation is limited by the hardware it runs on. Should this hypothesis prove true, it would place serious constraints on the practicality of obfuscation in most situations.

2 Background

This section outlines the definitions of boolean circuits and indistinguishability obfuscation that are essential to this paper.

2.1 Boolean Circuits

Real world programs are compiled into a sequence of machine instructions that can be interpreted by a processing unit (CPU) manipulating a finite number of *bits* at a time. The CPU is a chip that is made of logical gates that are made of a few transistors. In this context, boolean circuits are a powerful model that can express any algorithm we can run on a CPU[4].

Complexity Theory defines a *Boolean Circuit* C with n inputs and m outputs as a finite, labelled, directed acyclic graph. It contains n nodes with no incoming edges or ancestors; called the input nodes and m nodes with no outgoing edges or successors, called the output nodes. All other nodes are called gates and represent the logic operation AND, OR and NOT[2]. Simply put, a *Boolean Circuit* is a diagram showing a combination of logic operators to a drive an output from an input. By associating a boolean function with each gate, a boolean circuit computes an arbitrary function $f \in F : \{0,1\}^n \to \{0,1\}^m$. Two common properties are associated with boolean circuits:

1. The *size* of a circuit is the number of non-input vertices that it contains.

2. The *depth* of a circuit is the length of the longest directed path from an input vertex to an output vertex.

Because of the acyclic and oriented nature of the circuit, each node can be represented as receiving its inputs from a lower order depth only. To evaluate a circuit, we process the nodes in an increasing order of depth by

applying the associated boolean operation on the input vertices. The result is passed to the higher order nodes.

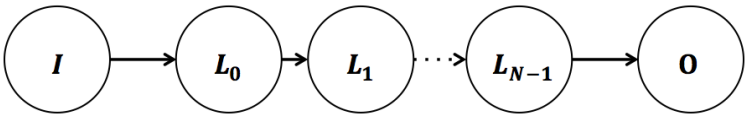

Figure 1: Simple high level representation of a boolean circuit

Size and Depth give an indication of the computation necessary to evaluate a circuit. Size is a rough measure of the computation time a single processor would need for a sequential circuit evaluation. Depth is an important cost metric for time efficiency in a parallel execution environment where the size is no longer the main bottleneck[10].

Another established circuit property that is relevant to our study is the *width* of a circuit[9]. If the circuit is arranged in levels such that the input of each level are connected from lower order levels exclusively, we define the width w as the maximum number of gates contained in single level.

The formal study of circuit complexity is beyond the scope of this paper. We should however quickly mention the notion of *Circuit Family*. Our definition limits the size of input to a fixed number of bits. The notion of boolean circuit C_n computing f_n is generalised to a family of circuits C which is a collection of boolean circuits $\{C_1, C_2, C_3, \cdots\}$ such that there is one circuit C_i that computes F on i inputs.

2.2 Indistinguishability Obfuscation

Obfuscation must meet two requirements, as defined by Barak et al.[11]:

1. The functionality of the obfuscated programme must remain identical to the functionality of the original, un-obfuscated programme.

2. Anything (and everything) that can be efficiently computed from Oc can be efficiently computed given oracle access to C. i.e. Oc does not disclose any information about the input other than what C does.

The first requirement is trivial and means that for any input programme C, an Obfuscator O produces a new programme O_C where the functionality of O_C is identical to the functionality of C. This condition means that given an input space I we have $\forall x \in I, C(x) = O_C(x)$.

The second requirement is less trivial and also known as the Virtual Black Box (VBB) requirement. It implies that there is no "leakage" of information, other than the input and the output. This means that it is impossible to tell whether an output was produced from the computation of

C, or from the computation of O_C. An adversary can learn a predicate of $O_C(x)$ with the same probability of learning it from $C(x)$.

Barak et al. concluded that the VBB requirement is impossible to attain in certain situations[11], however, it is possible to come close to meeting the second requirement, which in practice is enough to maintain the usefulness of obfuscation as an effective security measure. This is done by modifying the requirement of VBB, and replacing it with two weaker definitions that can still result in rendering a programme unintelligible in a meaningful and practical way. These definitions are namely indistinguishability iO and differing-input obfuscation diO, which require that for any circuit pair C_1 and C_2 from the same circuit family C that compute the same functionality f, the obfuscated versions of C_1 and C_2; $iO(C_1)$ and $iO(C_2)$ are perfectly indistinguishable. The probability of an observer being able to tell if an output was generated from $iO(C_1)$ or from $iO(C_2)$ is negligible.

Some work has been done within the scientific community to develop real-life obfuscations that meet the requirements given by Barak et al. Garg et al.[6] published an obfuscation construction valid for all polynomial-size, log-depth circuits constructed under novel algebraic hardness assumptions. Based on this work, Banescu et al.[3] published a proof-of-concept implementation of this iO construction. However, to date there is no efficient practical implementation of Barak et al.[11] indistinguishability obfuscation requirement. The study of the security of functional obfuscators is beyond the scope of this paper.

But this lack of ready-made obfuscators does not limit the ability of this paper to prove or disprove the hypothesis that obfuscation is ultimately limited by the hardware restraints of the operating system, as a simulation of an obfuscation-like process is sufficient for this purpose.

3 Methodology

We now describe the algorithm used to simulated an obfuscated boolean circuit and the tools used to measure its evaluation impact on a hardware platform.

3.1 BPW Generation

Thomborson proposed a space-efficient and computationally-appropriate file format (BPW) for the evaluation of very large Boolean functions with bounded program widths[9]. Gates of a circuit are represented by descriptors holding information about their type (AND, OR, ...) and are sequentially encoded inside the file's body. In our experiment we generate random boolean circuits (which evaluates a random boolean function) with varying width $w = \{10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$ for a fixed circuit size $N = 10^8$. The rela-

tionship between a circuit width and depth is $N = w \times D$, so for $w = N$ the circuit has depth 1.

The generated circuits are represented using a simplified BPW format by storing the gate type and the input indices only and by making some assumptions about the properties of the circuit under test. In particular, we make the following assumptions about the size of the circuit's inputs and outputs as well as the manner outputs from level L_i are passed to gates in level L_{i+1} as inputs.

1. All circuits have size $N = 10^8$
2. A circuit input array has size w.
3. Any given level L has exactly w gates.
4. Every gate has only one output.
5. A circuit output array has size w.
6. Any given gate from a level L_i has between one and three inputs that come from level L_{i-1} exclusively, such that: $G_i = f(X)$, where f is a boolean operator and X is an 8-byte array of size 1, 2 or 3.
7. The output of a gate is written inside the output array at the same index of the gate. $G_i(X) = Output\ array[i]$.

The *rand()* method from the *C* standard library was used to attribute a random type to a gate, as well as assign its input indices.

3.2 Evaluation

A circuit evaluation is sequential at both gate and layer level. During execution, gates of a level L_i are processed in ascending order of index in the level. Inputs indices for a gate are randomly distributed over [0, w-1]. Every gate retrieves its relevant inputs as described by the gate descriptor. Once a gate is processed, its output is stored in the output array at the same index as the gate. Once all the gates of level L_i are processed, the execution cursor moves to level L_{i+1}. The random distribution of gate input indices will significantly affect the overall performance of the circuit for large w, as the entire input array cannot fit in the cache. Figure 2 describes how a level is evaluated.

3.3 Hardware

Obfuscators might occasionally run on super computers, however it is likely that viable commercial use of obfuscators would require them to run on

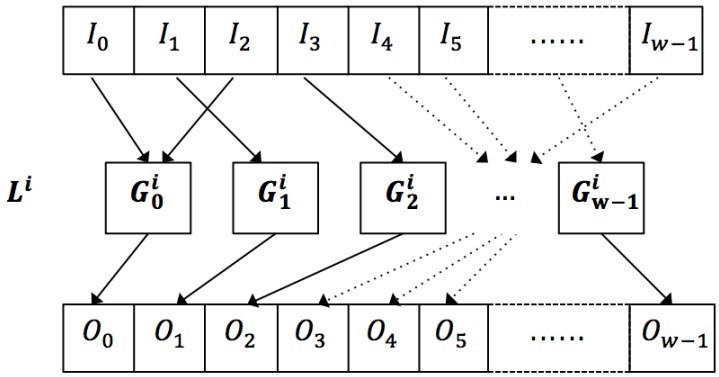

Figure 2: Simple description of Level L_i evaluation

mass market platforms such as smart phones, battery powered laptops and desktop computers[9]. The target platform for the circuits evaluation is a mid-level desktop computer. The CPU is a 64-bits-instruction-set with a base frequency of 3.40 GHz (Max of 4.1GHz). It has 4 cores that can run a thread each and has the following cache capacities (in bytes):

- Level 1 data cache: 32K
- Level 1 instruction cache: 32K
- Level 2 cache: 256K
- Level 3 cache: 6144K

We predict that having a circuit with a width higher than the capacity of the last-level cache is a perfect candidate to observe an impact on performance.

The operating system used is Linux Fedora 22 on kernel 4.4.5.

3.4 Metrics

The *perf* linux command was used to measure the impact of evaluating a circuit on the CPU. *perf* is a tool written in C that uses Performance Counters to profile an application[1]. Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted[8]. The *perf* tool has the advantage of being program oriented, instead of system oriented [1]. In particular, the *perf stat* subcommand allows to retrieval event counts that are of interest to our experiment. These metrics are mainly *instructions, cycles* and *LLC-loads*.

- **instructions** are unitary operations executed by the CPU at the lowest level such ADD, SUB, LOAD or STORE. They can represent arithmetic operations, data handling operations or control flow operations.

[1] `https://github.com/torvalds/linux/tree/master/tools/perf`

Modern CPUs include more complex instructions for complicated integer or float point operations.

- **cycles** are the steps performed by the CPU when it recieves an instruction. Typically early processors would pipeline fetching the instruction from the program counter, decoding it, reading its associated variables from the registers, executing its logic order and finally writing the result back to registers or storing it the system's main memory[7]. After an initial set up time, pipeling allows to execute one instruction per cycle, assuming no stalls. Modern CPUs achieve more than on instruction per cycle using techniques like *Instruction-level Parallelism* (loading more than one instruction in the pipeline) and *out-of-order execution* (allowing the simultanious exectution of multiple instructions that don't depend on each other for input or output). The Intel Core i5 can handle up to 4 instructions per clock cycle[5], but this figure is usually lower because of stalls, i.e. when the pipeline is waiting for the result of a previous instruction before proceeding.

- **LLC-loads** are instructions to load results from the Last-Level of Cache (or L3) of the CPU. L3 cache is the largest and highest-latency level of cache for the Intel Core i5 CPU. It is shared amongst all its cores.

For every value of w, we configure *perf* to run the experiment 3 times and collect the arithmetic average of the event counters.

4 Simulation Results

In this section we present the results of our experiment.

We suggest that the total running time of a circuit follows the analytical model:

$$T = \max\{t_g + t_e, t_m(w)\} \tag{1}$$

where T is the total elapsed (wallclock) time in processing a single gate, t_g is the CPU time for generating a gate, t_e is the CPU time for evaluating a gate, and $t_m(w)$ are the memory-stalls from the gate-generation and gate-evaluation computations.

In our experiment, we generate the entire circuit prior to its evaluation. t_g is in fact related to the time cost of loading of the gate for execution. We assume that this time component is near constant per gate and is not affected by the width w. The generation step executed initially is therefore not monitored for performance. The resulting circuit is stacked gate by gate in a simplified BPW file. Following the initial BPW format developed by Thomborson, the first 4 bytes represent the header of the a BPW file[9]. The following bytes then represent the total gates of the circuit. All gates

w	10^3	10^4	10^5
task-clock (ms)	13116.554712	13322.905953	13313.335011
instructions	99200423097	97602445259	97553743526
cycles	55393862507	56255744084	56207517444
instruction/cycle	1.74	1.76	1.74
branches	17179168771	16896966649	16887492840
branch-misses	213275235	267635214	267596585
LLC-loads	75319029	75177952	78311853
std deviation	1.49%	0.94%	0.27%

Table 1: Simulation results for $w \in \{10^3, 10^4, 10^5\}$

are sequentially represented from index 0 to index $10^8 - 1$. A given gate's encoding varies from 9 to 25 bytes. The first byte holds the type of the gate. The following 8 to 24 bytes present the input indices encoded over 64-bits each. The gate's representation could be optimised to the minimum bytes needed to encode a gate. Compression could also be used to save disk space. We chose to keep the existing model for simplicity. As mentioned earlier, the size of the circuits under test was fixed to $N = 10^8$. The average size of a BPW file storing a circuit is therefore $avg(sizeof(Gate)) * 10^8 \approx 2GB$ [2].

w	10^6	10^7	10^8
task-clock (ms)	13633.97763	16041.15891	17507.07431
instructions	97578653791	98201779911	103646369628
cycles	57584456723	67736311966	73941409511
instruction/cycle	1.69	1.44	1.4
branches	16897296695	17058902603	18501020880
branch-misses	266970610	266910904	268274635
LLC-loads	328139283	380853849	505923626
std deviation	0.43%	0.50%	0.19%

Table 2: Simulation results for $w \in \{10^6, 10^7, 10^8\}$

From the results collected in tables 1 and 2 we established the following:

$$T = \begin{cases} t_g + t_e & \text{if } \log_{10}(w) < 7 \\ t_m(w) & \text{if } \log_{10}(w) > 7 \end{cases}$$

We estimate that $t_m(w)$ for $w = 10^3$ is negligible. In our experiment $E(w = 10^6)$ the total run time is assumed to reflect load and execution

[2]We modeled 14 types of Gates; 1 × 1-input, 6 × 2-input and 7 × 3-input gates. On average, a gate has 2.43 inputs encoded with 8 bytes per input plus one byte for the gate type.

operations only (with no width penalty) which establishes $t_g + t_e \approx 130$ ns.[3]. Since the CPU has a frequency of 3.40 GHz the load and execution of a gate require about 450 CPU cycles per gate[4], which is up to 2000 instruction per cycle (at 1.7 instructions per cycle) on a quad-core CPU assuming no stalls. For all tested $w < 10^7$ we observed a stable instruction per cycle rate at 1.7 ins/cycle with 0.2 instruction per cycle stalled.

For $w > 10^7$ the computation becomes memory-bound, allowing us to estimate $t_m(10^7) = 162$ns and $t_m(10^8) = 176$ns. Since we used 6 2-input gate types and 7 3-input gate types (out of 14 gate types), a gate evaluation will have about 2.5 memory-fetches on average to read its inputs. The gate evaluation process will also write the result in the output buffer which will cause additional memory writes. There will also be some memory-reads when reading the gate's type.

Once the gate is unpacked in a top-level C **struc** of 16 bytes (1 byte for the type, 8 bytes for the input pointer, before padding to the largest power of 2), with 2.5 input indices of 8 bytes each, a level needs $(16 + 2.5 \times 8) \times w$ of memory to be fully represented, e.g. levels for $w = 10^7$ need 3.6 GB. This amount of memory cannot obviously reside in cache, and on 4 GB memory platforms there would certainly be page-faults requiring hard disk activity. We estimate that the level array can fit inside our system's 16GB RAM without disk faults. Our initial algorithm loads an entire level before feeding into the execution loop. We could improve this by evaluating a gate as soon as it is loaded.

We designed the gate inputs to be spread randomly in an array of width w. For any large enough w, the CPU last level cache (LLC) will miss with a high probability when the input bit-array cannot be fully contained. On our test platform, LLC is 6MB wide.

For w=1e5, the L3 cache occupancy of this array is still pretty high, at 1e5/6e6 = 16.6%; so whenever a (portion of a) gate-descriptor is pre-fetched into a line of L3 there won't be any CPU stalls on the pre-fetch but there's a 16.6% chance of evicting a line from the input array.

For $w = 10^7$, the input array occupies 8×10^7 bits, the miss rate should be $1 - (6 \times 10^6 B / 10^7 B) = 40\%$. From the observerd $T = 162ns$ we deduce the memory latency, which is the time required by the CPU to fetch the data from the cache, to be $162ns$ per LLC miss.[5]. For $w = 10^8$, the expected miss rate is 94% and the memory latency is 75ns.

[3] For $w = 10^3, N = 10^8$, we have $N \times T = 13116$ms $\implies T \approx 130$ns

[4] $130 \times 10^{-9} \times 3.4 \times 10^9 = 442$

[5] $latency = \frac{T}{2.5 \times \text{miss rate}}$ where T is the total run time per gate

5 Conclusions and Future Work

Obfuscation is a potentially powerful tool to encrypt and protect data in the future. However, one of the limitations of obfuscation may prove to be that it lowers the execution speed of a programme, at least in some circumstances.

In this experiment we simulated an obfuscated circuit, and evaluated the performance of that circuit. We have shown how perf hardware counters can be used to measure cache hits inside a 6MB L3-cache ix86 microprocessor clocked at 3GHz in order to evaluate a simulated obfuscated circuit.

Based on the execution times observed in the experiment, we have proposed an analytical model to explain the execution times. This analytical model holds true even when the circuit width is large, i.e. where $w > 10^6$. Further work is required to determine whether this analytical model can predict execution speed to a statistically significant degree of accuracy. A more extensive study, using a large range of hardware devices and over a large range of circuit simulations, could help validate or disprove the proposed model.

We plan to improve our simulation by replacing the random generation mechanism using the C random function with the stride method described in this paper. We also plan to use the Intel Performance Counter Monitor library as a high level interface for measuring real time performance data directly inside the code. In particular, we plan to measure the CPU cache and instruction metrics at initialisation time, then at generation time and finally at execution time. This will help make life better for everyone.

References

[1] R. Arnold. A. Zanella. *Evaluate performance for Linux on POWER. Analyze performance using Linux tools*. 2012. URL: `http://www.ibm.com/developerworks/linux/library/l-evaluatelinuxonpower/`.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009, pp. 101–103. URL: `http://theory.cs.princeton.edu/complexity/circuitschap.pdf`.

[3] Sebastian Banescu et al. "Idea: Benchmarking Indistinguishability Obfuscation – A Candidate Implementation". English. In: *Engineering Secure Software and Systems*. Ed. by Frank Piessens, Juan Caballero, and Nataliia Bielova. Vol. 8978. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 149–156. ISBN: 978-3-319-15617-0. DOI: `10.1007/978-3-319-15618-7_12`. URL: `http://dx.doi.org/10.1007/978-3-319-15618-7_12`.

[4] Stephan Mertens Cristopher Moore. *The Nature of Computation*. OUP Oxford, 2011, p. 131.

[5] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers.* 2016. URL: http://www.agner.org/optimize/microarchitecture.pdf.

[6] Sanjam Garg et al. *Candidate Indistinguishability Obfuscation and Functional Encryption for all circuits.* Cryptology ePrint Archive, Report 2013/451. 2013.

[7] John Morris. *Computer Architecture.* SOFTENG363, University of Auckland. 2008. URL: https://www.cs.auckland.ac.nz/~jmor159/363/html/pipelines.html.

[8] Perf. *Linux Kernal Organisation - Perf Wiki.* 2015. URL: http://perf.wiki.kernal.org.

[9] Clark Thomborson. "Benchmarking Obfuscators of Functionality". In: *CoRR* abs/1501.02885 (2015). URL: http://arxiv.org/abs/1501.02885.

[10] Ryan Willams. *An Overview of Circuit Complexity.* URL: http://web.stanford.edu/~rrwill/week1.pdf.

[11] B. Barak O. Goldreich R.Impagliazzo S. Rudich A. Sahai S. Vadhan K. Yang. *On the (Im)possibility of Obfuscating Programs.* 2001.