The University of Auckland

COMPSCI 780

Benchmarking software obfuscators
An experimental approach

Rafik Nasraoui

supervised by
Dr. Clark Thomborson

May 8, 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 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 definitons 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[1].

Complexity Theory defines a *Boolean Circuit* C with n inputs and m outputs as a finite, labeled, 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.

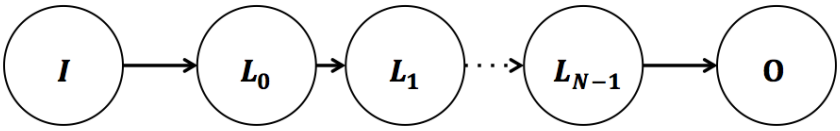


Figure 1: Simple high level representation of a boolean circuit

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[3].

Another established circuit property that is relevant to our study is the *width* of a circuit[4]. 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.[5]:

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

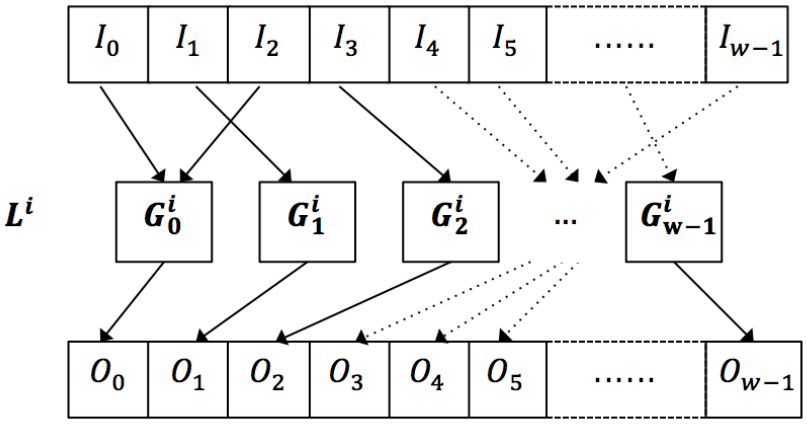
C, or from the computation of O_C. An adversary can learn a predicate of $O_C(x)$ with the same probability of learning it from $C(x)$.

Barak et al. concluded that the VBB requirement is impossible to attain in certain situations[5], 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.[7] published a proof-of-concept implementation of this iO construction. However, to date there is no efficient practical implementation of Barak et al.[5] indistinguishability obfuscation requirement. This lack of ready-made obfuscators does not limit the ability of this paper to prove or disprove the hypothesis that obfuscation is 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 assimilate a boolean circuit to an acyclic directed graph and suggest an experiment to benchmark its evaluation.



3.1 Assumptions

In order to simulate a boolean circuit and estimate its evaluation, we make some assumptions about its properties. In particular, we make the following assumptions about the circuit's inputs and outputs as well as the manner outputs from level L_i are passed to gates of level L_{i+1} as inputs.

1. Any given level L has at most w gates.
2. The total number of inputs I is equal to the total number of outputs O and satisfies the following: $w < I < 3 * w$
3. Any given gate from level L_i has between one and three inputs that come from level L_{i-1} exclusively. We can write the following: $G_i = f(X_{i-1})|$ f is a bool operator $, X_{i-1} = x^1_{i-1}||x^1, x^2_{i-1}||x^1, x^2, x^3_{i-1}$
4. Every gate has one output and one output only.
5. The output of gate is written inside the output array at the some index of the gate. $G_i(X) = Output\ array[i]$.

We also assume that this time component is constant per gate. The generation step is executed initially and is not monitored for performace. The resulting circuit is stacked gate by gate in a BPW file. Following the initial BPW model **??** the first 4 bytes of the a BPW file correspond to the "magic caracteres" B- P- W- 1-. The following bytes represent the total gates of the circuit. Assuming the total number of gates $N = 10^8$, all gates are sequentially stacked from indice 0 to $10^8 - 1$. A given gate's encoding varies from 5 to 13 bytes. The first byte holds the type of the gate. The following 4 to 12 bytes present the input indices encoded over 64-bits each. We could optimise the gate's repsentation further but we kept this representation for simplicity. For $N = 10^8$ the average test BPW file is

$$avg(sizeof(Gate)) \times 10^8 \approx 800MB \tag{1}$$

The entire content of a BPW file is initially loaded in memory using a array of type Byte (64-bits). The array is then read byte-by-byte to reconstruct the circuit. Once a gate is reconstructed, it gets evaluated by retrieving its corresponding inputs from the inputs array. The resulting ouput is written to the ouputs array at the same index of the gate.

Our approach uses hardware counters to obtain the values of instructions and cycles run as well as the misses at first and last level of CPU cache.

3.2 Measurement

The results collected are shown in Figure x.

We have run the benchmark 6 times for a fixed total number of gates but a variable width w in $10^3, 10^4, 10^5, 10^6, 10^7, 10^8$. We designed the benchmark

to reveal whether the size of w will have a critical effect on performance. Our reasoning is as follow:

T = Ta + Tb + Tw

If Tw is too small (define too small) then Tw is negligible comparing to Ta and Tb. However, starting from a particular threshold, Ta + Tb become negligible relatively to Tw. Equation x summarises our analysis of the performance of our benchmark:

Eq here

From the results collected, we can observe that when $w < 10^7$, $t_m(w) < t_g + t_e = 130$ ns and when $w >= 10^7$, the computation becomes memory-bound, allowing us to estimate $t_m(10^7) = 162$ nsec and $t_m(10^8) = 176$ nsec.

3.3 Generation And Execution Time

The $t_g + t_e = 130$ nsec estimate seems reasonable: it is about 45 CPU clocks. Assuming no stalls, a dual-issue CPU clocked at xxGHz can execute up to 90 instructions during that time. Our testing plateform has an x86 instruction and the observed issue rate is relatively stable at 1.7 inst/clock for all tested $w < 10^7$. This is not a surprisingly-low (nor a surprisingly-high) issue-rate for a dual-issue i586. We could also estimate the instructions per gate at 1.7*45 = 75 instructions.

3.4 Memory Time

Regarding $t_m(w)$: about half of your gates are 3-input and half are 2-input, so there'll be about 2.5 memory-fetches per gate-eval (to read its inputs). There'll also be some memory-writes during the gate-evaluation process (when you're unpacking the gate-descriptors), some memory-reads (when you're reading the unpacked gate-descriptions); and you'll be writing the gate-output bits. You *might*

Your gates are unpacked into a top-level struct of 16 bytes (after padding to the next-larger power of two), and each gate references 1 to 3 input-descriptors of 8 bytes each (these are packed in the "inputs" array). At 2.5 inputs/gate that's 20 bytes for the input-descriptors plus 16 bytes for the top-level gate-descriptor – a total of 36 Bytes per gate, or 36e8 = 3.6 Gbytes per level when $w = 10^8$. It's certainly not cache-resident and if you had only 4 GB of main memory there'd be some "hard" page-faults requiring disk activity. But you're probably (just) ok here; once again it'd be a good idea to avoid unpacking a gate-descriptor from a BPW file until you're ready to evaluate it, rather than unpacking an entire level...

The gate input-reads are scattered more or less randomly through an array of w bits – so they're almost-always an LLC miss for any w large enough that this bit-array doesn't fit in your 6MB = 48 Mb LLC, i.e. whenever $w > 10^7$. The miss rate should be very close to (48e6/1e8) =

48% for w=1e8. With 1 gate eval every 176 nsec, and (2.5)(48%) = 1.2 misses/gate-eval, that's a memory-latency of 176/1.2 = 147 nsec per LLC miss. That's in range of the (indicative) latency data I see in point 4 of https://software.intel.com/en-us/articles/intelr-memory-latency-checker – and your memory is pretty heavily loaded for bandwidth, from these gate-input reads and also from the gate-descriptors.

So... the model seems plausible... however we'd have a significantly-lower miss rate if your gate input-reads aren't actually scattered randomly through an array of w bits, due to a defect in your random-number generation.

As for the possibility that your reads and writes of unpacked gate-descriptors are causing a memory-bandwidth bottleneck... each level of a $w = 10^8$ circuit takes 36e8 bytes to store its gates in unpacked format. That's a total of 2*36e8B/17.6s = 410 MB/s in memory bandwidth. Your performance monitor reports "28.817 M/sec" for the LLC-load rate. I have *no idea* whether these are 32B loads or 16B loads or 64B loads – but if they're 32B loads then we get 922 MB/s. That leaves 510 MB/s for the gate-input reads, at 32B per LLC miss. Above I estimated one LLC miss on the gate-input reads every 147 nsec: that's 217 MB/s. Dunno what's happening for the other 300 MB/s, but there will be some additional traffic, so it's an ok explanation even though it'd be good to have some story to tell about this 300 MB/s of "other" memory bandwidth...

If each LLC load is 16B, then your perf-counts are reporting 461 MB/s of memory bandwidth, leaving only 51 MB/s for the gate-input reads. At 16B per LLC miss, these add up to about 108 MB/s – so if an LLC load is 16B the data is not well-explained by the analysis above.

I have to go now... I hope this made some sense! The idea is for you to tell some story, along the above lines, about your observations. If you're unable to explain some aspects of your data, that's fine, you're certainly not expected to be an expert in performance-modelling of memory systems! Instead you should "keep it simple" in your explanations and raise some questions (in your analysis) which merit further study.

4 Conclusions and Future Work

We have shown how perf hardware counters can be used to measure cache hits inside a 6MB L3-cache ix86 microprocessor clocked at 3GHz evaluating a simulated obfuscated circuit. We have proposed an analytical model to explain the execution times observed. Even for large width circuits where $w >> 10^6$, ... We plan to improve our simulation by replacing the random generation mechanism from using the C random function to using a '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 mea-

sure the CPU cache and instruction metrics at initialisation time, then at generation time and finally at execution time.

References

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

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[3] Ryan Willams. An overview of circuit complexity.

[4] Clark Thomborson. Benchmarking obfuscators of functionality. *CoRR*, abs/1501.02885, 2015.

[5] B. Barak O. Goldreich R.Impagliazzo S. Rudich A. Sahai S. Vadhan K. Yang. On the (im)possibility of obfuscating programs, 2001.

[6] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. Cryptology ePrint Archive, Report 2013/451, 2013.

[7] Sebastian Banescu, Martín Ochoa, Nils Kunze, and Alexander Pretschner. Idea: Benchmarking indistinguishability obfuscation – a candidate implementation. In Frank Piessens, Juan Caballero, and Nataliia Bielova, editors, *Engineering Secure Software and Systems*, volume 8978 of *Lecture Notes in Computer Science*, pages 149–156. Springer International Publishing, 2015.