OBFUSCATION THROUGH THE OBSERVER-EFFECT: THINKING OUTSIDE THE VIRTUAL BLACK-BOX

By

Jeremy Lee Blackthorne

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

_____

Bülent Yener, Thesis Adviser

_____

Ana Milanova, Member

_____

Boleslaw K. Szymanski, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2015
(For Graduation May 2015)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my advisor Professor Bülent Yener for his mentorship and always allowing me to make the mistakes I needed while preventing me from wasting time on mistakes I did not. I would like to thank the other two members of my thesis committee, Professor Ana Milanova and Professor Boleslaw Szmanski, for granting me there time. I would like to especially thank Benjamin Kaiser for many, many hours of fruitful collaboration regarding this research. I would like to thank the senior members of RPISEC, Markus Gaasedelen, Sophia D'Antoine, Alexei Bulazel, Patrick Biernat, Austin Ralls, and Brandon Clark for their persistent willingness to discuss my security research. I would like to thank the members of Group 59 of Lincoln Laboratory for their guidance and support. Finally, I would like to thank my wife for her patience and love.

ABSTRACT

Theoretical investigations of obfuscation have been built around a model of a single Turing machine which interacts with a user. A drawback of this model is that it cannot account for the most common approach to obfuscation used by malware, the observer-effect. The observer-effect describes the situation in which the act of observing something changes it. Malware implements the observer-effect by detecting and acting on changes in its environment caused by user observation.

In this work, we initiate a theoretical study of obfuscation with regards to programs that interact with a user and an environment. We define the System-Interaction model to formally represent this additional dimension of interaction. We also define a *semantically obfuscated* program within our model as one that hides all semantic predicates from a computationally bounded adversary. This is possible while still remaining useful because semantically obfuscated programs can interact with an operating system while showing nothing to the user. Next, we analyze the necessary and sufficient conditions of achieving this standard of obfuscation. Finally, we demonstrate a candidate approach to achieving those conditions on current computers.

CHAPTER 1
Introduction

Program obfuscation is the transformation of code with the intent of making it "hard" to understand while maintaining functionality. Authors of commercial software obfuscate their products to protect their intellectual property, criminals obfuscate their malware to protect against detection by anti-virus software, and nations obfuscate cyber-weapons to prevent repurposing. But no obfuscation technique thus far has been able to guarantee any provable security for everyday programs. This is partly due to the large divide in the theoretical and systems approaches. Each approach has its strengths and weaknesses. It is the goal of this work to combine the strengths of both to shed additional light on the subject of obfuscation.

Theorectical Approach. Theoretical work in obfuscation focuses on simplified models, provable results, and well-defined properties. The most famous of these properties is the virtual black-box (VBB) property, and was defined by Barak *et al.* [1]. Informally, the property states that an adversary should gain no more information from the source code of the obfuscated program than they would gain by having black-box access to the original program. This is seen as the ultimate goal of formalized obfuscation because no more information could possibly be hidden without also hindering functionality. Recently, VBB obfuscation schemes were constructed in [2] and [3].

Another standard of obfuscation is defined by the indistinguishability property, also first established in [1]. It concerns functionally equivalent programs that have multiple distinct circuit implementations. The property states that such a program could be run without the adversary knowing which of the possible circuits was actually used. In [4], this construction was proven secure under the colored matrix model, which only allows adversaries to perform computations on matrices in a specific order.

The are many other well-defined obfuscation types, such as extractability [5],

virtual grey-box [6], tau [7], and best-possible [8]. All of these obfuscation types are weaker than VBB, so all comparisons to our own obfuscation scheme will be with VBB.

Limitations. A common theme among theoretical obfuscation is the use of a single Turing machine (TM) as a model. By this we mean the program to be obfuscated is a single TM which interacts with a user. This model leads to natural limits in obfuscation. For instance, if the program is to be useful in any way, it must allow access to its input and output. This limits the goal of obfuscation to protect only the transformation from input to output, while still allowing the adversary or user access to the inputs and outputs. Consider the case of malware, in which a program needs to interact with a system and simultaneously not leak any information to the user. The single TM model does not account for interactions other than with the user, and hence cannot properly represent this case.

Another limitation implicit in the single TM model is the lack of security possible for learnable functions. A function is learnable if an adversary can determine the definition of the function through only its inputs and outputs. Consider a program that implements the function $f(x) = x^2$. Even without access to the program itself, an adversary can guess the definition by querying the function at a few locations. Because programs obfuscated in the single TM model must allow access to inputs and outputs to be useful, they cannot be both learnable and meaningfully obfuscated.

Systems Approach. The systems approach relies on the concept that observation *requires* modification. This modification can be either of the program itself or the environment. Because observation requires modification, and modification can be detected, observation can be detected. Authors can design programs that detect observation and create deviations in execution, which ultimately impedes observation. This allows obfuscated software to behave in different ways, depending on whether or not it is being observed.

The set of techniques that leverage the observer-effect are known most commonly by their anti-X names, where X is the observation environment. Examples are

anti-debugging [9][10][11], anti-virtual machine (VM) [12][11], and anti-emulator [13] [14]. Observation environments like debuggers, VMs, and emulators tend to have small differences from a normal execution environment. These differences are known as artifacts. The anti-X techniques look for artifacts during execution and change the behavior of the program when they are found. One goal of these analysis environments is to not create artifacts. The result is an arms race between detecting new artifacts and building environments that do not have them. In theoretical obfuscation, conversely, the adversary is allowed to observe without cost, i.e. there are no artifacts and there is no effect on the running program when it is being observed.

Limitations. The major drawback of the system approach is the lack of provable security. Many obfuscation schemes have been proposed and implemented using the methods in the systems approach, but ultimately are heuristic with no provable security [15] [16]. Security standards cannot be proven because the concepts are not sufficiently formalized.

1.1 Results

System-Interaction Model We introduce a model that accounts for the hardware, operating system (OS), program, and user. We allow the hardware to have access to a unique random oracle based on its own physical phenomena. A user submits the OS and program to the hardware for computation. The hardware returns the results to the user and to the OS, based on the OS and program that was submitted to it.

We define a new type of obfuscation within this model called semantic obfuscation. This obfuscation hides all semantic predicates from a user while still remaining useful. This is possible within the System-Interaction model because a program can still give output to the OS while showing nothing to the user.

Existing Sensors Two common ways a program can measure itself and its environment is memory hashing and timing checks. Memory hashing describes the process of a program running a hash over sections of memory. Timing checks are

the recording of the time it takes to execute batches of instructions. Both of these can be used to check the integrity of a program or its environment. We formalize these sensors and strengthen them by allowing them access to the random oracle of the hardware. We show an impossibility of achieving semantic obfuscation with either of these sensors individually or combined.

Ideal sensor The ideal sensor returns a random number based on the state of the OS and program. In this way, no bit can be changed in either without it registering with the ideal sensor. We show that with access to a ideal sensor, there exists a semantically obfuscated program that runs in polynomial time and which guarantees exponential time to deobfuscate. The catch is that is also takes exponential time to construct. We show that a semantically obfuscated program which uses a ideal sensor actually can take no less than exponential time to construct. This obfuscator must create the obfuscated program using the hardware on which the obfuscated program is intended to run because each hardware has its own unique random oracle. This naturally places the obfuscator at a practical disadvantage because the adversary can try to deobfuscate the program with the hardware of their choice, including much faster hardware than what was intended.

1.2 Related Work

In addition to the fields of obfuscation and malware analysis, our work relates to tamper-resistance. Canetti and Varia formally define nonmalleable obfuscation, another term for tamper-resistance, and show how existing point functions can achieve their security standard in the random oracle and common reference string models [17]. Another formalization of tamper-resistance is provided by Basile et. al [18], for which they define tamper protection techniques in terms of attacker goals, capabilities, and limitations.

A hardware approach to tamper-resistance is physically unclonable functions (PUF). These are hardware devices that are unique, hard to replicate, and produce random output. The concept of program sensors within the System-Interaction model is closely related to work done in PUFs. Recent work in PUFs has explored

the idea of using intrinsic properties of commercial off-the-shelf systems [19] [20]. Plaga and Koob [21] formally describe PUFs and their limitations.

CHAPTER 2
Preliminaries

2.1 Notation

TM is short for Turing machine. UTM is short for universal Turing machine. PPT is short for probabilistic polynomial-time Turing machine. A TM can also be encoded as a bitstring $b \in \{0,1\}^n$ and used as input to other TM's. Oracle access is input-output only access. A TM A running on input string x with oracle access to a TM B is represented as $A^B(x)$. A function $\alpha : \mathbb{N} \to \mathbb{N}$ is called negligible if it grows slower than any other polynomial.

2.2 Definitions

Definition 2.2.1. A Turing machine [22] is defined by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are finite sets and

1. Q is the set of states,
2. Σ is the input alphabet without the blank symbol $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Definition 2.2.2. (Instantaneous Description). The instantaneous description (ID) of a TM is defined as $ID\colon TM \to \Gamma^*, M \mapsto \gamma$, where

1. M is some TM,
2. and γ is the contents of the tape at the time-step in which the ID function is called. The contents of the tape include M and any input to M.

Definition 2.2.3. Learnable Function [23]
A function f, computed by a Turing machine M, is learnable if there exists a PPT L such that

$$Pr[X \leftarrow L^M(1^{|M|}) : X = M] \geq \alpha(|M|)$$

Informally, a function is learnable if an adversary can query that function at a finite number of locations and extract the definition of that function.

Definition 2.2.4. Point Function
A point function is one which returns false on all inputs except for one, for which it returns true. An example function:

$$f(x) = \begin{cases} False : x \neq 37 \\ True : x = 37 \end{cases}$$

A practical example of a point function in computation is a password authentication function. This would return false on all inputs except the password.

We use the notation M_k to indicate a TM M that returns false on all inputs x except when $x = k$. And similarly we use the notation f_k to indicate the corresponding function that M_k is implementing.

We use the point function as the simplest possible function in our proofs for which to demonstrate properties of sensors.

Definition 2.2.5. (Virtual Black-Box Obfuscation) [**?**]. A PPT $\mathcal{O}$ is a virtual black-box obfuscator if the following conditions hold:

1. *Functionality:* $\mathcal{O}(M)$ and M are input/output equivalent.

2. *Polynomial Slowdown:* $\mathcal{O}(M)$ is at most polynomial longer in running time and polynomially larger in description length than M. Specifically there exist a polynomial p such that for every TM M, $|\mathcal{O}(M)| \leq p(|M|)$ and if M halts in t steps on input x, then $\mathcal{O}(M)$ halts in at most $p(t)$ steps on x.

3. *Virtual Black-Box Property:* For all semantic predicates π:

$$Pr[A(\mathcal{O}(M)) = \pi(M)] - Pr[S^M(1^{|M|}) = \pi(M)] \leq \alpha(|M|)$$

2.3 Properties of Turing Machines

A property of a TM can be expressed as a yes-no question, sometimes called a predicate. An individual predicate of a TM is denoted by $\pi(M)$, where M is any TM. The set $\Pi = \{\pi_0, \pi_1, \pi_2, \ldots\}$, is the set of all predicates over a TM. This is the set of Turing decidable languages and thus infinite.

There are two types of TM properties, *semantic* and *syntactic* [24]. Semantic properties are those that are dependent on the input-output relationship of the function the TM implements. In other words, a semantic property of a TM is also a property of the language which the TM recognizes. Syntactic properties are not necessarily dependent on the language which a TM recognizes, but rather the encoding of that TM. Consider a program that loops x times and prints "hello world" each time. We consider the output to be what is printed on the screen, namely "hello world" x times. An example of a syntactic change of this program would be if we changed the program so that it consists of two loops which iterate half the times. This change would not change any semantic properties of the TM because we have not changed the input-output relationship of the function. If we changed the original program so that it printed "hello world" x^2 times instead of x times, then we would have made a semantic change because we have changed the input-output relationship of the function.

2.4 Obfuscation

Obfuscation is the syntactic transformation of a program while maintaining its semantic properties. In other words, the obfuscated program and original program should be input-ouput equivalent, yet appear completely different. It is the relations between the syntax and semantics that are being obfuscated.

CHAPTER 3
System-Interaction Model

The model we outline in this chapter is conceptually simple, but allows for a real type of obfuscation yet to be formalized. We begin by explaining the assumptions that motivated this model's construction.

3.1 Assumptions

To decide properties of TMs, we must first observe them. We know that with real software, program states are mostly hidden. For every line of text, or single animation shown, a program may be executing thousands of instructions with many variables. To gain information about these hidden program states, we have two options; We can observe the internal states directly by simulating the program in an analysis environment or we can try to infer program state by what the program shows publicly. This is stated more formally in the following assumption:

Assumption 3.1.1. *(Observation). Given any two TMs A and B, the only ways for A to observe B's current configuration are directly by simulating B or indirectly through B's output.*

If simulating a program is not an option, we must restrict our observation to the program's output. If the program output does not give us the information we want, we can change the program to produce additional output. These changes can be made to the program itself or the system on which it runs. An example of direct modification would be inserting print statements into the program to leak internal state information. An example of modification of the system would be running a program in a debugger. A debugger does not actually modify the program, just the environment.

Assumption 3.1.2. *(Modification). The only ways to change a TM's output is by changing the TM itself or the UTM simulating it.*

When a real program is running on a computer, we know that the program can take its environment as input. An example of this is a program checking for how many other programs are currently running in the same environment. We call the mechanism that a program uses to read in its environment a sensor.

Assumption 3.1.3. *(Sensation) A TM A has the ability to read information about a TM B when B is simulating A.*

We combine these simple ideas into a single model called the System-Interaction model.

3.2 Definitions

Definition 3.2.1. (System-Interaction Model). The hardware H is a unique physically implemented two-tape UTM with access to a unique random oracle R based on physical phenomena. H is a black-box that takes input from the user which we represent by a PPT A. The user gives input in the form of an operating system U, program M, and any input x_1 to M. U is an encoding of a UTM and M is an encoding of a TM. H then simulates $U(M(x_1))$ and produces two outputs: (y_1, y_2). The first output y_1, will be returned to the user via being written to the user-tape. The second output y_2, is the output to the operating system U via the system-tape. The user can only see the output that is returned to it via the user-tape. In addition to user-input, H can give an input to M, labeled x_2. H would then simulate $U(M(x_1, x_2))$.

Definition 3.2.2. (Program). A program M is a two-tape TM that implements the following function:

$$M : \Sigma^* \times \Sigma^* \to \Gamma^* \times \Gamma^*$$

$$x_1, x_2 \mapsto y_1, y_2$$

where

1. x_1 is the user input string to M,
2. x_2 is the system input string to M,
3. y_1 is output from M written to the user-tape,
4. and y_2 is the output from M written to the system-tape.

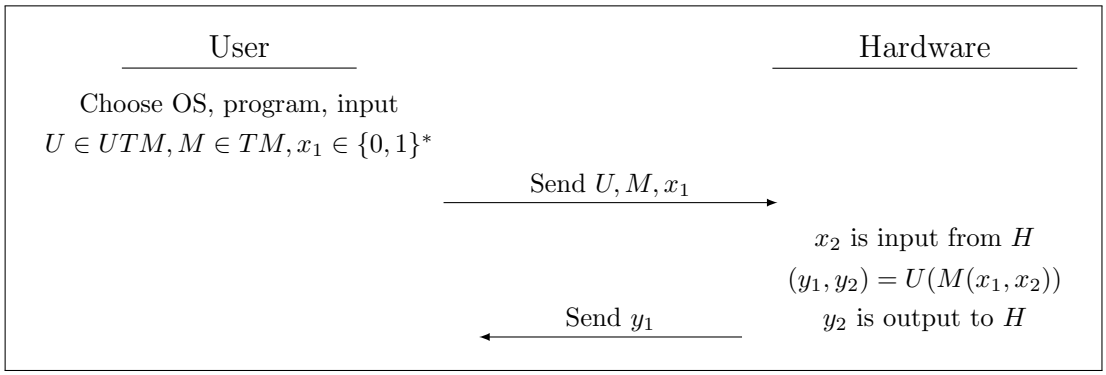


Figure 3.1: This illustrates basic interaction between user, program, OS, and hardware.

3.3 Adversaries

To more accurately represent the obfuscation of real programs, we have expanded the single TM model into the System-Interaction model. But in order to gain any insights from our work, we must limit the scope. We have chosen to not consider the hardware-based adversary, as in circuit level instrumentation. We have also chosen not to address the problem of information leakage through changes to the system. An example of this would be the adversary that inspects the state of

the system after the program runs. For this work, we assume that after the program has completed execution, the system U resets back to the state it held upon being loaded onto the hardware. The adversary is free to make any changes to U and then load it onto the hardware.

Given the limitations above, there are three natural approaches to determining a property of a program M: statically extracting information from M alone, emulating the system H, and changing U or M to leak information. We consider each in turn.

3.3.1 Static Analysis

Static analysis is the technique of analyzing a program without running it. This means that a system is not needed to simulate it. VBB obfuscation prevents any information leaking other than through the inputs and outputs of a program. This is half of the solution to leaking any information through M. The other half of the protecting M is through protecting its inputs and outputs.

3.3.2 Emulation

We consider the possibility of the hardware H being simulated by another software or hardware UTM H'. This would be akin to placing an operating system in a hypervisor or hardware emulator. Empirical results show that any H' could simulate H, but not with exact similarity to that of the original hardware H [25]. To reflect this, we say that H can be simulated on H', but the responses of R on H' will be randomly different from the responses of R on the original hardware H. Our assumption allows us to ignore the adversary which emulates the hardware.

Assumption 3.3.1. *H can be simulated on any UTM $H' \neq H$, but the responses of R' on H' will have no correlation to the responses of R on the original hardware H.*

3.3.3 Modification

The third approach is the modification of the operating system U and is the one on which we focus in this paper. If the program has no publicly available

input or output, then the adversary is prevented from indirect observation. Because of Assumption 3.3.1 regarding simulation, the only possibility left is changing the program or the system to leak information.

3.4 Semantic Obfuscation

This new model of obfuscation calls for an exploration of obfuscation ideals. In the single TM model, virtual black-box obfuscation was the absolute ideal because no more information could be hidden without also hindering functionality. In our model with multiple observers, namely the user and the system, a program can hide all semantic information from the user while still remaining useful by interacting with the system.

Definition 3.4.1. (Semantic Obfuscation). A TM $\mathcal{O}$ is a semantic obfuscator if the following conditions hold:

1. *Functionality:* $\mathcal{O}(M)$ and M are input/output equivalent.

2. *Polynomial Slowdown:* $\mathcal{O}(M)$ is at most polynomial longer in running time and polynomially larger in description length than M. Specifically there exist a polynomial p such that for every TM M, $|\mathcal{O}(M)| \leq p(|M|)$ and if M halts in t steps on input x, then $\mathcal{O}(M)$ halts in at most $p(t)$ steps on x.

3. *Semantic Security:* $\mathcal{O}(M)$ hides all semantic properties from an adversary bounded polynomially in time and space.

CHAPTER 4
Sensors

Programs can measure properties of themselves and the systems on which they run. We call these types of measurement functions sensors. In our System-Interaction model, we label the system input $x_2 = H()$ as a sensor and label it $x_2 = sensor()$ when that value depends on the environment. An example would be $x_2 = sensor(U, M)$. This would return different values based on the values of U and M.

4.1 Learnable Sensor

It is trivial to show that semantic obfuscation cannot be achieved in the System-Interaction model when the sensor is a learnable function. Polynomial number of queries by the adversary is enough to forge the sensor, making the hardware unnecessary to run the program correctly.

Theorem 4.1.1. *An author cannot create a semantically obfuscated program within the System-Interaction model when the sensor is learnable.*

Proof. Let M_k have no user input or output and write some number y_2 to the system tape when given the system input $x_2 = k$. Let $k = sensor(U, M_k)$, where $sensor : U \times M \to \mathbb{N}$ and $sensor$ is learnable.

The adversary creates U' which prints the system output y_2 to the user tape. The adversary slightly modifies U' and repeats the process n times, where n is polynomial with respect to the size of M. He can then derive the function f which the sensor computes. This allows him to compute any value of the sensor without running on the intended hardware H, including $k = sensor(U, M_k)$. This allows the adversary to simulate the hardware and compute $y_2 = M_k(k)$, which allows the derivation of the function computed by M_k. This allows the adversary to compute semantic properties of M_k, thus violating the property of semantic security guaranteed by semantic obfuscation.

□

4.2 Random Oracle Sensor

Opposite of a learnable function is an unlearnable function or one in which the definition of the function cannot be determined through inputs and outputs alone. An example of an unlearnable function is a random oracle. A random oracle is a function which returns a unique, perfectly random value for every input. Given the same input, a random oracle will return the same random value. A random oracle can be thought of as the most unlearnable function because an infinite number of inputs and outputs will not give any information about any other input-output pairs.

We now model a sensor that is a random oracle. This sensor allows a program to detect any differences made in the program or its environment. With access to this ideal sensor, we can now achieve semantic obfuscation.

Theorem 4.2.1. *There exists a semantic obfuscator within the System-Interaction model when the sensor is a random oracle.*

Proof. We will construct an obfuscator $\mathcal{O}$ in the System-Interaction model with random oracle sensor such that $M_k = \mathcal{O}(U, M_k)$, $k = sensor(U, M)$ on H, M has no user input or output, M_k is VBB obfuscated and calls M with no user output when receiving an input of k, otherwise M_k outputs a 0 to the user and halts.

To construct M_k, we wrap M in a point function such that at the point k, M_k will call M, all other points M_k just halts. For $i = 0$ to $i = n$, we construct M_i, with the goal of $i = sensor(U, M_i) \bmod 2^n$, where n is our security parameter. This gives us the ability to increase or decrease the codomain of the random oracle. We construct 2^n different M_i's, VBB obfuscate each one, and submit them to H. The single M_i that returns no user output will be the program for which $i = sensor(U, M_i)$. Now that we have constructed M_k, we must show that M_k is semantically secure. To do so we employ a semantic security game.

We allow a PPT adversary A to pick any two programs of equal size such that M_1, M_2, such that neither have user input or output, and send them to the challenger. The challenger will flip a fair coin to choose a bit b. The challenger then computes $M_k = \mathcal{O}(U, M_b)$ and sends back M_k to the adversary. The adversary must determine whether M_k is the obfuscated version of M_1 or M_2.

The adversary has three sources from which to extract information from M_k: the user output, source code, and the system output. By definition there is no user output from the program. Our obfuscator is a VBB obfuscator so the source code does not leak any information. This leaves the only source of information to be the system output. In the System-Interaction model, the user cannot see the system output without modifying U. If the adversary runs M_k with a modified U', the reading from $sensor(M_k, U')$ returns some x not equal to and independent of k, which is needed by M_k to call M and produce the correct system output. M_k is a point function with a domain of 2^n, which means it is computationally infeasible for a PPT adversary to guess k. □

Discussion. We have shown that we can create a semantically obfuscated program but the obvious drawback is that it was done in exponential time. In the world of cryptography, exponential time is reserved for the adversary, not the legitimate user. In the case obfuscation, it could still be considered useful. This is because even though the creation time is exponential, the deobfuscation time would be exponential, while the running time of the obfuscated program would be polynomial.

Theorem 4.2.2. *No polynomial time semantic obfuscator $\mathcal{O}$ exists within the System-Interaction model when the sensor is a random oracle.*

Proof. Assume by way of contradiction that $\mathcal{O}$ is polynomial time obfuscator in the System-Interaction model with ideal sensor such that $M_k = \mathcal{O}(U, M)$, where $k = sensor(U, M)$. M_k is VBB obfuscated and only runs M when receiving an input of k, otherwise M_k outputs a 0 to the user and halts.

If PPT $\mathcal{O}$ can transform M into M_k, then there exists a dependent relationship between M and M_k by way of the algorithm $\mathcal{O}$, which implies a dependent relationship between $x = sensor(U, M)$ and $k = sensor(U, M_k)$. This is a contradiction because $sensor()$ is a random oracle, guaranteeing total independence of outputs given any two different inputs.

□

Discussion. We have highlighted the essential limitation of using the observer-effect. The strongest observer-effect possible, formalized here in the ideal sensor, necessarily limits the creation of the program as much as the deobfuscation.

Theorem 4.2.3. *A semantic obfuscator $\mathcal{O}$ cannot obfuscate a program M in less time than an adversary can deobfuscate $\mathcal{O}(M)$ within the System-Interaction model when the sensor is a random oracle.*

Proof. As proven in Theorem 4.2.2, an author of any semantically obfuscated program M_k in the System-Interaction model with random oracle sensor can at best create M_k in exponential time. This computational effort is also restricted by the speed at which H runs. This is because the System-Interaction model assumes that H is unique and independent for all $H' \neq H$. This forces the author to generate 2^n programs and test each on H.

By default, the adversary will not see any output from M_k when running on H with U. Thus, the adversary must change U to U' to see the system input and output. This necessarily changes $x = sensor(U', M_k)$ and forces the adversary to brute-force all possible $U' \neq U$ such that $sensor(U', M_k) \bmod 2^n = sensor(U, M_k) \bmod 2^n$, where n is the security parameter. The adversary can parallelize these brute-force attempts over any number of H's. Thus not being subject to the bottleneck of running on H and H alone, as the author does. This gives the adversary a large computational time advantage.

□

Discussion. Theorem 4.2.2 showed that the observer-effect guarantees a computational symmetry between obfuscation and deobfuscation within the System-Interaction model. But Theorem 4.2.3 highlights an additional consideration in the real world, hardware speed. The obfuscator extracts secret information from H and embeds it in the program. But this new representation of the secret knowledge has an innate vulnerability, in that it allows itself to be extracted by the adversary on faster hardware than what the original obfuscator could do. This shows that not only can the observer-effect not be used to any advantage by the obfuscator but it

is actually to the obfuscator's disadvantage.

4.3 Piecewise Learnable Sensor

We established in Theorem 4.1.1 that a learnable sensor cannot be used to build a semantically obfuscated program. We proved in Theorem 4.2.2 that a sensor that functions as a random oracle cannot be used to create a semantically obfuscated program in polynomial time. We are now left with unleanable functions which are not random oracles as possible candidates for a semantic obfuscator that runs in polynomial time.

We now model a sensor as an unlearnable function which is a hybrid of both learnable functions and a random oracle. The sensor is *piecewise learnable*. This means the sensor function as a whole is not learnable, but each sub-function by itself is learnable.

The total state space of the system is $S = U \times M$, with the system being in any given state $S_t \in S$ at time t. The state of the system S_t is a binary encoding of a TM and so can be sectioned off by S_i, where $S_i + 1$ is some arbitrary number, and $i \in \{0, 1, \ldots, n-1, n\}$. There are $n = 2^\lambda$ number of different learnable sub-functions. The assignment of each sub-function to a subset of the state space is unknown a priori, but is itself measurable. We model this by assigning each sub-function to a subset of the state space by a random oracle R.

$$Sensor(S_t) = \begin{cases} f_{R(0)}(S_t): & S_0 \leq S_t < S_1 \\ f_{R(1)}(S_t): & S_1 + 1 \leq S_t < S_2 \\ \qquad \ldots & \\ f_{R(n)}(S_t): & S_{n-1} \leq S_t \leq S_n \end{cases}$$

With this sensor, it is not clear how it can be used to create a semantically obfuscated program. We need to be able to make changes to the program to embed the correct k such that $k = sensor(U, M_k)$. Simultaneously, we need the adversary

to not be able to make changes to the program. These two conditions can be summarized with the following two properties of our piece-wise learnable sensor:

1. $S'_t = (U, M_{k'})$ and $S_t = (U, M_k)$ fall under the same sub-function. And because each sub-function is learnable, there exists a PPT A such that given $x_2 = sensor(U, M'_k)$, A can derive $x_2 = sensor(U, M_k)$.

2. $S'_t = (U', M_k)$ and $S_t = (U, M_k)$ do NOT fall under the same sub-function. This means there does NOT exist a PPT A such that given access to $x_2 = sensor(U', M_{k'})$ can derive $x_2 = sensor(U, M_k)$.

If the sensor is learnable then condition 1 is true and 2 is false. If the sensor is a random oracle then 1 is false and 2 is true. It it clear we need both 1 and 2 to be true. Let us now prove that having both conditions true is sufficient to create a semantic obfuscator in the System-Interaction model.

Theorem 4.3.1. *Given a sensor which can be represented by a piecewise learnable function, there exists a PPT $\mathcal{O}$ that transforms a TM M into a semantically obfuscated program M_k, within the System-Interaction model.*

Proof. Let M be a program with no user input or output. We construct a program M_k such that $k = sensor(U, M_k)$. We do this construction in constant time through public-key encryption. This is achieved by creating M_i from $i = 0$ to $i = p$, where p is polynomial with regards to the size of M. Each M_i writes out $x_2 = sensor(U, M_i)$ to the user tape. The author can decrypt these readings with the private key. Given access to p sensor readings, all from the same sub-function, the author computes M_k such that $k = sensor(U, M_k)$ in that sub-function.

The adversary is required to use instrumentation to determine the input/output of the program because they do not possess the private key. And due to property two of our piece-wise learnable sensor, changing U to U' will cause the sensor to jump sub-functions. The new sub-function and its sensor values will be completely independent of the original sub-function the program was in when it was being run with U, thus giving the adversary no information. □

Discussion. Given the hefty assumptions we made, it does not come as surprise that we can achieve semantic obfuscation. It does help illustrate a point though: We need an asymmetry in the observer-effect for it to be useful. Quite in like cryptography, we need a one-way function. One-way functions in cryptography are over the domain of integers or lattices. Our one-way function is over the domain of TM's in the System-Interaction model.This comes in the form of the assumptions that changing keys in our point function doesn't change the sub-function, but changing the environment does change the sub-function.

CHAPTER 5
Existing Sensors

So far we have considered properties of functions that act as sensors. We have shown that it is necessary that a function be unlearnable but not a random oracle. Then we showed that a piece-wise learnable function is sufficient to achieve semantic obfuscation within the System-Interaction model.

In this chapter we will consider existing sensors within real programs and show what relation they have to the functions we have already considered.

5.1 Static Sensor

The simplest type of sensor in real programs is the the self-memory check. A program can run a hash over its own memory to determine if anything has been changed. We represent this capability in our model with the following assumption:

Definition 5.1.1. (Static Sensor). Any program M has oracle access to a Sensor(S), where Sensor(S) = $R(S)$, R is the random oracle from the hardware H, $S \subseteq ID(U(M))$.

We show why this simple type of sensor cannot be used to ensure integrity for obfuscation.

Lemma 5.1.1. *An adversary can leak the return value of sensor*(s) *in the System-Interaction model.*

Proof. We begin by changing U to write out the entire contents of the system tape to the user-tape just before any call to Sensor(S). The tape contents D are then modified to remove the changes made to U. The modified version of D will be denoted D'. Then a new program M' is constructed with the old tape contents appended as data D'.The new program M' contains D', and makes a call to Sensor(S), where S is now equal to subset of the tape that contains D'. The call to this sensor will return the same value as the sensor call in the original program M. This shows

that the return value of the static sensor, even when measuring any subset of the system, cannot be hidden. □

5.2 Dynamic Sensor

Programs can also measure properties that change over time. A practical example is the use of the RDTSC x86 assembly instruction which measures an internal hardware clock. Pairs of these instructions can be used to measure the time it takes for code to execute between them. We formalize the idea of measurement over time by first introducing the notion of a trace.

Definition 5.2.1. (Trace). The trace (Tr) of a TM M is defined as the ordered set of IDs of M from timestep$= 0 \dots t-1$, where t is the current timestep.

To represent a simple timing based sensor like RDTSC, we introduce a dynamic sensor which sums all the values in a trace.

Definition 5.2.2. (Dynamic Sensor). Any program M has oracle access to Sensor(Tr), defined as $\text{Sensor}(Tr) = \sum_{i=0}^{t-1} R(ID_i)) \mid ID_i \in Tr(U(M))$, and t is the current time-step.

The dynamic sensor is also inadequate for use in ensuring integrity for obfuscation purposes.

Lemma 5.2.1. *An adversary can leak the return value of any call to Sensor(Tr) in a program M in the System-Interaction model.*

Proof. Leaking the value of Sensor(Tr) in a program M is trivial, simply because Sensor(Tr) does not measure any instructions that occur after the call to the sensor. To leak the value, modify the part of U that occurs after the call to $sensor(Tr)$ to write the result of the sensor call to the user-tape. Multiple calls to the sensor could be leaked in turn. For each call to Sensor(Tr), generate a new program which writes the return value of that call to Sensor(Tr) to the user-tape. □

5.3 Static and Dynamic Sensors

Real programs often combine static and dynamic sensors. A practical example is a program that does both hashes over its memory and checks the time it takes to compute those hashes. Even this combination of sensors cannot ensure integrity for obfuscation.

Theorem 5.3.1. *Semantic obfuscation is not possible with combined static and dynamic sensors when the dynamic sensor is learnable.*

Proof. The following is a simple algorithm to extract all of the sensor readings: First apply Lemma 5.1.1 to all calls to Sensor(S). Next we must extract the calls to Sensor(Tr). There are two possible cases.

Case 1: We can modify U to write the result of Sensor(Tr) to the user-tape without any call to Sensor(S) being affected. If this is true, we can apply Lemma 5.2.1 and are finished.

Case 2: There exists a call to Sensor(S) that will measure any modification of U needed to write the value of Sensor(Tr) to the user-tape. In this case, Sensor(Tr) will be affected by the modified return value of Sensor(S). But because Sensor(Tr) is linear and thus learnable, we can determine what the return value of Sensor(Tr) should be through summing a series of calls to Sensor(S), where S is set to areas of the tape that contain the intended ID(U(M)) for that timestep. □

Discussion. The underlying reason of this impossibility is the same as why a learnable sensor in general cannot be used. It is easy to see that this same impossibility applies when the dynamic sensor is a random oracle. Both cases reduce to the cases discussed in Chapter 4.

Now we will consider a construction of combined static and dynamic sensors, but this time we will assume the dynamic sensor is the piece-wise learnable sensor described in Chapter 4.

Theorem 5.3.2. *Given a static sensor and piece-wise learnable dynamic sensor, there exists a semantic obfuscator within the System-Interaction model.*

Proof. Let M be a program with no user input or output in the System-Interaction model. Let M have access to $sensor(S)$, which is a random oracle. Let M have access to $sensor(Tr)$ which is piece-wise learnable. We construct the program $M_k = \mathcal{O}(M, U)$, which runs M on input k. The obfuscator also VBB obfuscates M_k. We construct M_k in the same method as Theorem 4.3.1, but this time so it first calls $sensor(S)$ where $S = U(M_k)$ and then calls $x_2 = sensor(Tr)$. The program M_k then checks to see if x_2 is equal to k.

We consider M_k and M to be input-output equivalent because after unlocking M_k with k, M_k then calls M. The wrapper M_k calls M in constant time, so we may say that there is at most polynomial slowdown. Finally, we must establish the semantic security property.

The adversary can extract a semantic predicate from the source of M_k or modify U to print out additional information about M_k. We know that M_k is VBB obfuscated so no information can be attained from the source that can not also be attained from running M_k. The adversary can modify U to print out the value of x_2 by copying it to the user tape. If any modifications are made to U, then the value x_2 will change and become independent of the original value. This value x_2' will be random and independent from x_2 thus leaking no information. The program M_k will only exit upon running $M_k(x_2')$. Thus not allowing the adversary to determine any semantic predicates about M_k.

□

CHAPTER 6
Multithreading as Candidate Sensor

We know that a piece-wise learnable function is sufficient to achieve semantic obfuscation within the System-Interaction model. We also know that the weaker static and dynamic sensors are sufficient to achieve semantic obfuscation when the dynamic sensor is piece-wise learnable. We think that multithreaded race conditions could serve as this sensor.

Multithreading has a large impact on system dynamics and instruction ordering. When a program is multithreaded, the order in which the threads execute is external to the program itself. We assume statically determining order is nontrivial and that more realistically, it must be measured. We provide here the results of preliminary experiments demonstrating the feasibility of using multithreaded race conditions as a means to create a piece-wise learnable sensor.

6.1 Experimental Methodology

Our program consists of threads started in a loop, each overwriting a global variable with their thread ID. Once the variable has been assigned a thread ID, it is then encrypted with an RSA public key embedded in the program. This encrypted ID is then written to disk. After a thread completes the former actions, it exits. Once all threads have have completed, there will be a file on disk of the the n encrypted thread IDs, where n is the number of threads.

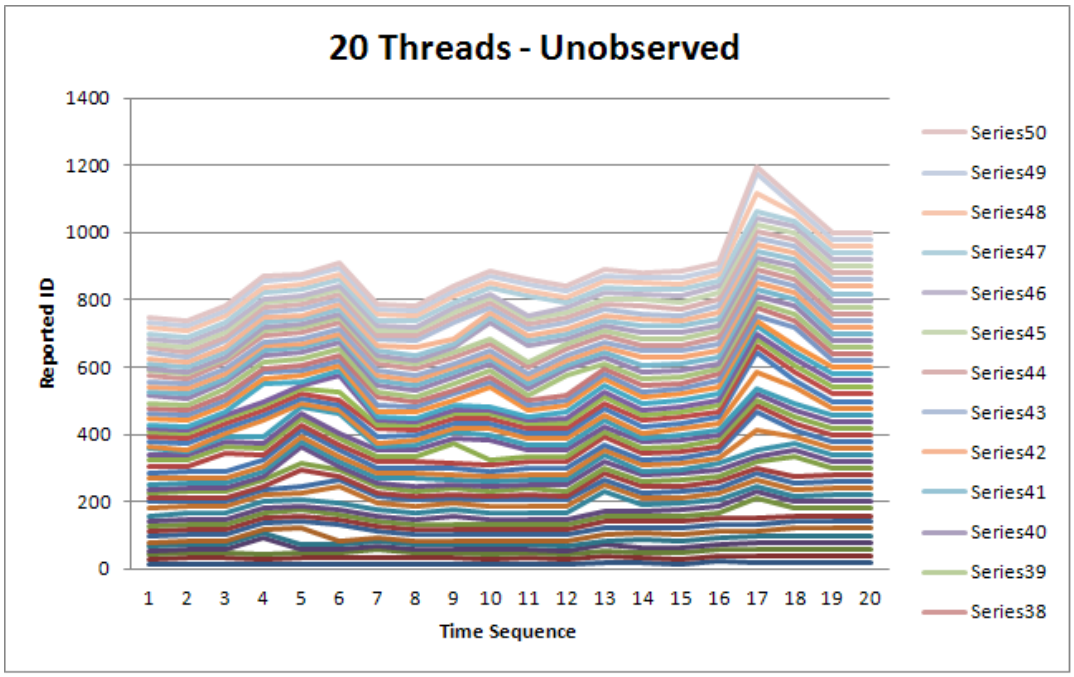
A person holding the corresponding private key, can then decrypt the information. An adversary without the private key, would be forced to modify the program or the environment to retrieve the information.

We completed this experiment using 20, 25, 30, and 35 threads. The experiment was repeated 50 times with each thread count. The experiments were completed on a MacBook Pro with a 2.7Ghz Intel Core i7 and 16GB 1600MHz of DDR3 RAM. The program was run on a Windows XP virtual machine within Virtual Box.

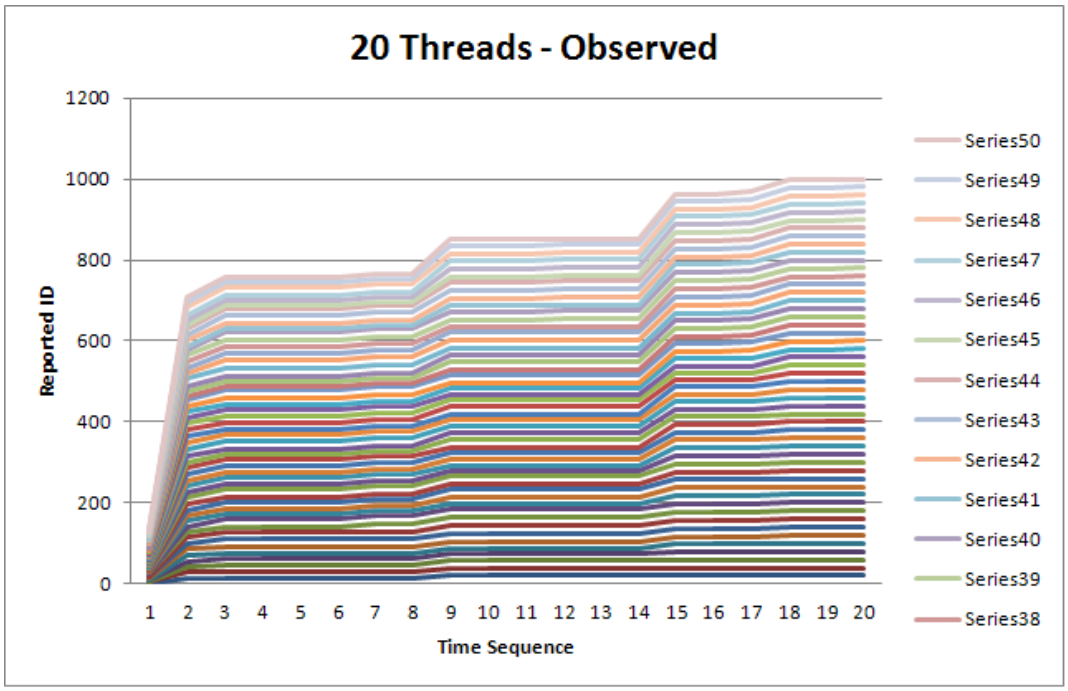
6.2 Results

The result of the experiment with 20 threads was a time series of 20 numbers. The experiment was repeated 50 times, so we have a matrix of 50 by 20. The results are similar for other thread counts with the 20 being replaced by the thread counts of 25, 30, and 35.

We graphed the results of each experiment set as a stacked line. This means that the values of the second experiment are added to the values of the previous experiment. We chose this format because it presents a clear contrast between the results of the observed and unobserved cases.

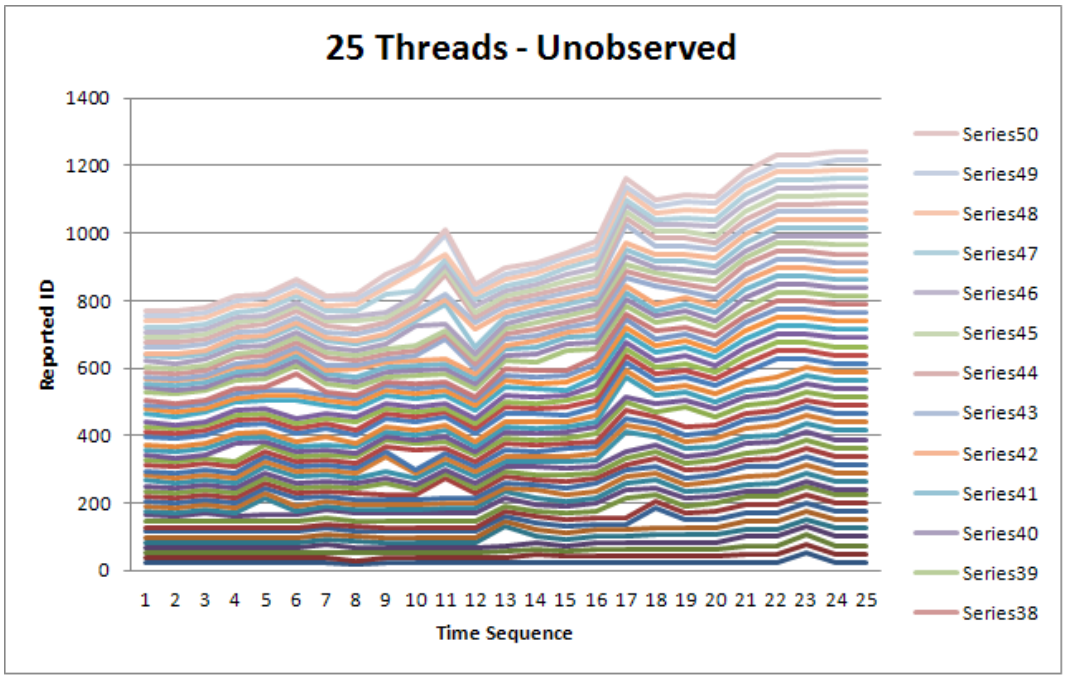


(a) Unobserved

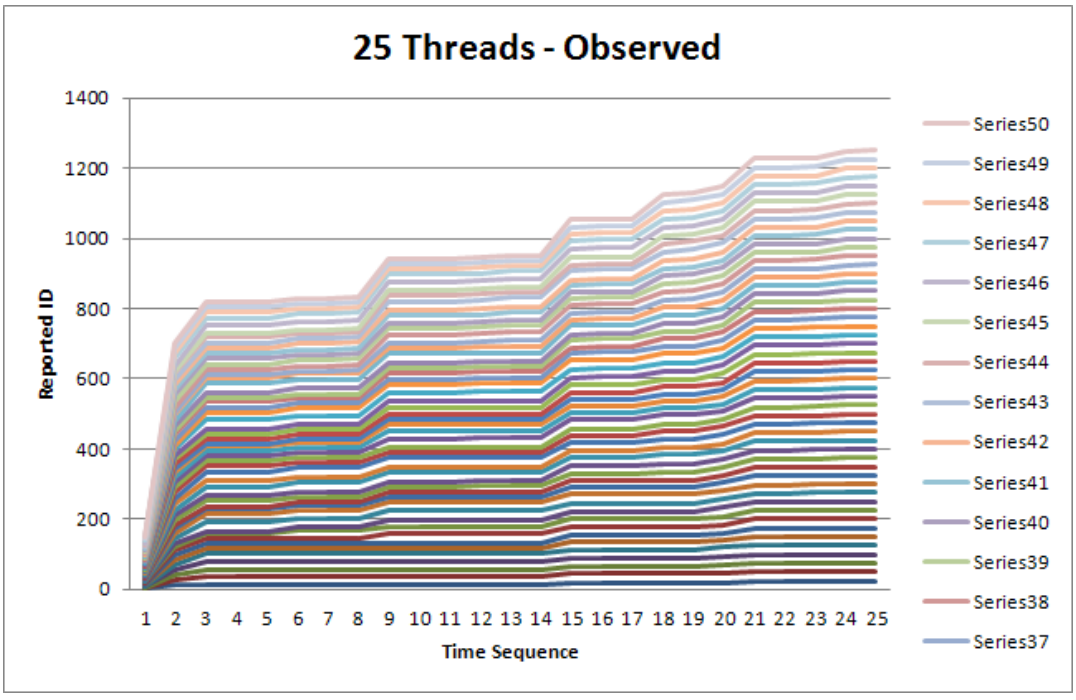


(b) Observed with printf

Figure 6.1: 20 Threads

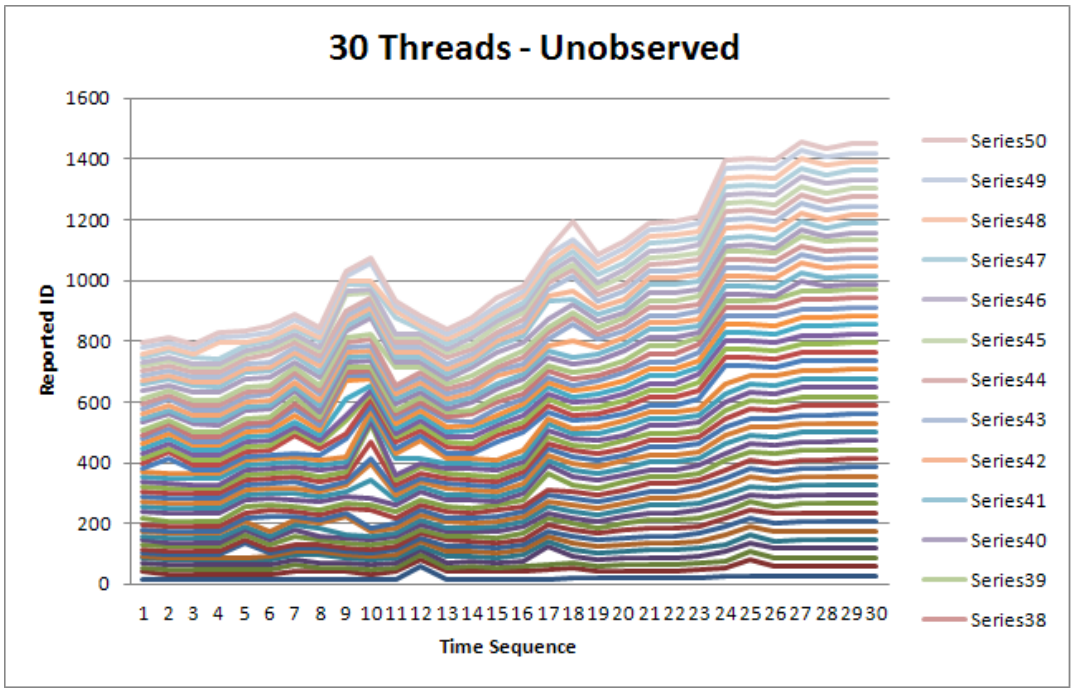


(a) Unobserved

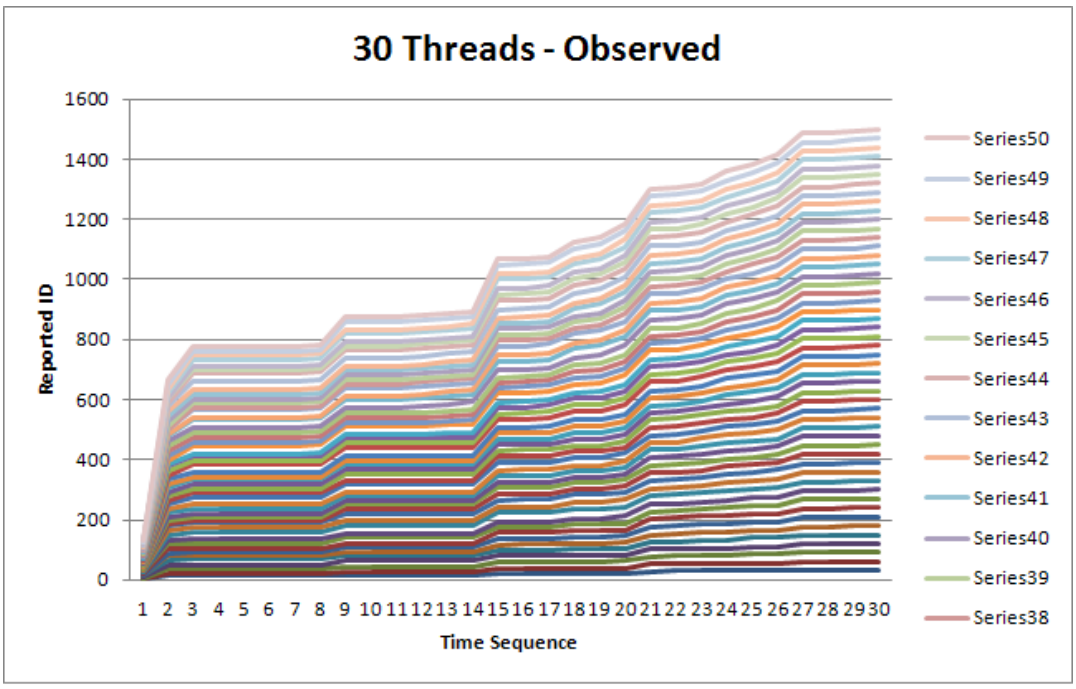


(b) Observed with printf

Figure 6.2: 25 Threads

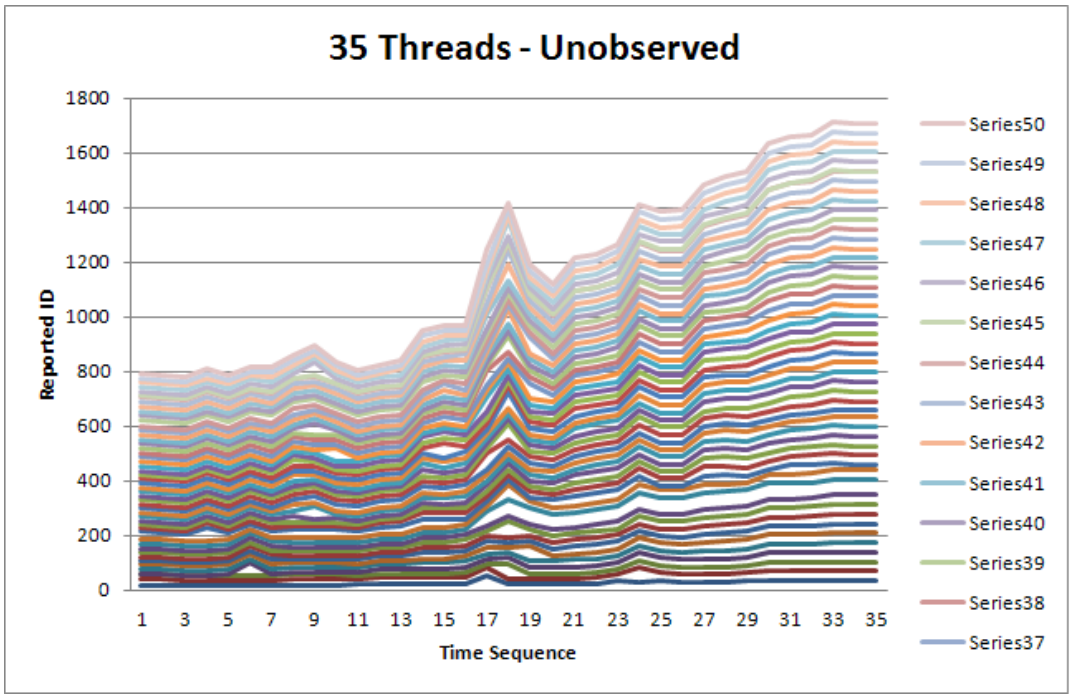


(a) Unobserved

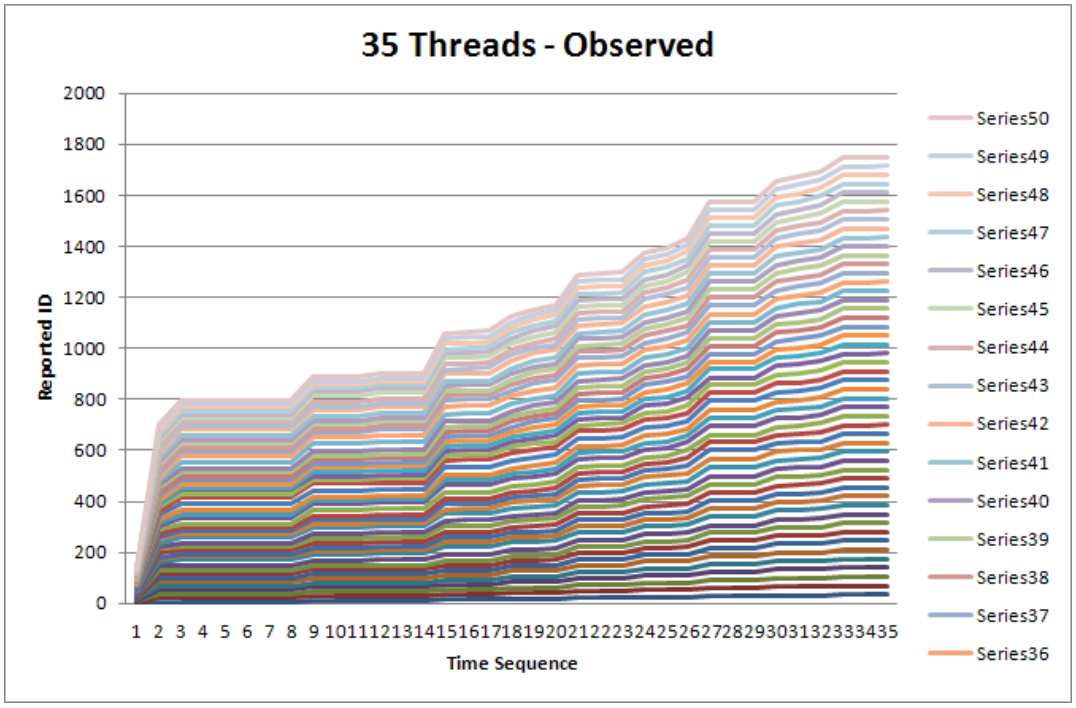


(b) Observed with printf

Figure 6.3: 30 Threads



(a) Unobserved



(b) Observed with printf

Figure 6.4: 35 Threads

6.3 Analysis

The graphs reveal starkly different trends between the observed and unobserved experiments. The stacked lines are actually demonstrated ordering. The observed cases report thread IDs in a nondecreasing order. The unobserved cases are mostly out of order.

Table 6.1: 50 Experiments Unobserved

Threads	Ordered	Unordered
20	6/50	44/50
25	2/50	48/50
30	2/50	48/50
35	6/50	4/50

Table 6.2: 50 Experiments Observed

Threads	Ordered	Unordered
20	50/50	0/50
25	50/50	0/50
30	50/50	0/50
35	50/50	0/50

The additional print statement added to the code caused the race conditions to change, thus causing different orderings. It also seems like the statement added just enough time so that the race condition disappeared, giving orderly behavior.

CHAPTER 7
Future Work

We have provided a formal framework for which to describe the observer-effect in programs. We constructed a well-defined standard of obfuscation within that framework, and we have shown the necessary and sufficient conditions of achieving that standard. We have even provided some rudimentary implementation and testing which demonstrates the feasibility of the formal framework.

All of this research has formed a basis for which to construct practical, semantically obfuscated programs. The clear next step, is to now do extensive empirical investigation into the nature of multithreading, concurrency, and possible other avenues for which to achieve semantic obfuscation.

REFERENCES

[1] B. Barak *et al.*, "On the (im)possibility of obfuscating programs," in *Proc. 21st Annu. Int. Cryptology Conf.*, 2001, pp. 1–18.

[2] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," Cryptology ePrint Archive, Report 2013/563, 2013. [Online]. Available: http://eprint.iacr.org/2013/563.pdf. Accessed Apr. 6, 2015.

[3] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai, "Protecting obfuscation against algebraic attacks," Cryptology ePrint Archive, Report 2013/631, 2013. [Online]. Available: http://eprint.iacr.org/2013/631.pdf. Accessed Apr. 6, 2015.

[4] S. Garg *et al.*, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *FOCS 2013*, pp. 40–49.

[5] G. Tan, Y. Chen, and M. Jakubowski, "Delayed and controlled failures in tamper-resistant software," in *8th Int. Workshop Information Hiding*, 2007, pp. 216–231.

[6] N. Bitansky and R. Canetti, "On strong simulation and composable point obfuscation," in *Proc. 30th Annu. Cryptology Conf.*, 2010, pp. 520–537.

[7] P. Beaucamps and E. Filiol, "On the possibility of practically obfuscating programs towards a unified perspective of code protection," *J. Computer Virology*, vol. 3, no. 1, pp. 3–21, Apr. 2007.

[8] S. Goldwasser and G. N. Rothblum, "On best-possible obfuscation," in *Proc. 4th Theory Cryptography Conf.*, 2007, pp. 194–213.

[9] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *IEEE Int. Conf. Dependable Systems and Networks*, 2008, pp. 177–186.

[10] P. Ferrie, "The ultimate anti-debugging reference," May 2011. [Online]. Available: http://pferrie.host22.com/papers/antidebug.pdf. Accessed Apr. 6, 2015.

[11] M. Sikorski and A. Honig, "Anti-debugging," in *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, San Francisco, CA, USA: No Starch Press, 2012, ch. 16, pp. 327–350.

[12] P. Ferrie, "Attacks on virtual machine emulators," in *Proc. of Assoc. of Anti-Virus Asia Researchers Conf.*, 2006, pp. 128–143.

[13] M. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proc. 1st ACM Workshop on Virtual Machine Security*, 2008, pp. 11–22.

[14] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proc. 3rd USENIX Conf. Offensive Technologies*, 2009, pp. 2–2.

[15] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. of Comp. Sci., The University of Auckland, Auckland, New Zealand, Rep. 148, 1997.

[16] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proc. 16th USENIX Security Symp.*, 2007, pp. 19:1–19:16.

[17] R. Canetti and M. Varia, "Non-malleable obfuscation," in *Proc. 6th Theory of Cryptography Conf.*, 2009, pp. 73–90.

[18] C. Basile and others., "Towards a formal model for software tamper resistance," COSIC, University of Leuven, Flanders, Belgium, 2009. [Online]. Available: https://www.cosic.esat.kuleuven.be/publications/article-1280.pdf. Accessed: Apr. 6, 2015.

[19] R. Nithyanand, R. Sion, and J. Solis, "Solving the software protection problem with intrinsic personal physical unclonable functions," Sandia Nat. Lab., Livermore, CA, USA. Rep. SAND2011-6603, 2011.

[20] R. Nithyanand and J. Solis, "A theoretical analysis: Physical unclonable functions and the software protection problem," in *Proc. 2012 IEEE Symp. Security and Privacy Workshop*, pp. 1–11.

[21] R. Plaga and F. Koob, "A formal definition and a new security mechanism of physical unclonable functions," in *Proc. 16th Int. GI/ITG Conf. Measurement, Modeling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, 2012, pp. 228–301.

[22] M. Sipser, "The Church-Turing thesis," in *Introduction to the Theory of Computation*, Independence, KY, USA: Cengage Learning, 2005, ch. 3, sec. 3.1, pp. 140–141.

[23] A. Saxena, B. Wyseur, and B. Preneel, "Towards security notions for white-box cryptography," in *Proc. 12th Int. Conf. Information Security*, 2009, pp. 49–58.

[24] S. Arora and B. Barak, "Randomized computation," in *Computational Complexity: A Modern Approach*, New York, NY, USA: Cambridge University Press, 2012, ch. 7, sec. 7.5.3, pp. 121–122.

[25] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities," in *Proc. 11th USENIX Workshop on Hot Topics in Operating Systems*, 2007, pp. 6:1–6:6.

APPENDIX A
Code for Experiments

A.1 Main Experiment Code

```
1  #define WIN32_LEAN_AND_MEAN
2  #include <windows.h>
3  #include <stdio.h>
4  #include <assert.h>
5  #include "stdint.h"
6  #include <stdlib.h>
7  #include <intrin.h>
8  #include <openssl/pem.h>
9  #include <openssl/ssl.h>
10 #include <openssl/rsa.h>
11 #include <openssl/evp.h>
12 #include <openssl/bio.h>
13 #include <openssl/err.h>
14
15
16 #pragma intrinsic(__rdtsc)
17
18 HANDLE thread1,thread2;
19 int id;
20 unsigned __int64 a,b;
21
22
23 DWORD WINAPI MyThreadFunction0( LPVOID lpParam );
24 DWORD WINAPI MyThreadFunction1( LPVOID lpParam );
25
26
27 char publicKey[]="-----BEGIN␣PUBLIC␣KEY-----\n"\
```

```
28 "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAy8Dbv8prpJ/0kKhlGeJY\n"\
29 "ozo2t60EG8L0561g13R29LvMR5hyvGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+\n"\
30 "vw1HocOAZtWK0z3r26uA8kQYOKX9Qt/DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQAp\n"\
31 "fc9jB9nTzphOgM4JiEYvlV8FLhg9yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68\n"\
32 "i6T4nNq7NWC+UNVjQHxNQMQMzU6lWCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoV\n"\
33 "PpY72+eVthKzpMeyHkBn7ciumk5qgLTEJAfWZpe4f4eFZj/Rc8Y8Jj2IS5kVPjUy\n"\
34 "wQIDAQAB\n"\
35 "-----END␣PUBLIC␣KEY-----\n";
36
37  char privateKey[]="-----BEGIN␣RSA␣PRIVATE␣KEY-----\n"\
38 "MIIEowIBAAKCAQEAy8Dbv8prpJ/0kKhlGeJYozo2t60EG8L0561g13R29LvMR5hy\n"\
39 "vGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+vw1HocOAZtWK0z3r26uA8kQYOKX9\n"\
40 "Qt/DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQApfc9jB9nTzphOgM4JiEYvlV8FLhg9\n"\
41 "yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68i6T4nNq7NWC+UNVjQHxNQMQMzU6l\n"\
42 "WCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoVPpY72+eVthKzpMeyHkBn7ciumk5q\n"\
43 "gLTEJAfWZpe4f4eFZj/Rc8Y8Jj2IS5kVPjUywQIDAQABAoIBADhg1u1Mv1hAAlX8\n"\
44 "omz1Gn2f4AAW2aos2cM5UDCNw1SYmj+9SRIkaxjRsE/C4o9sw1oxrg1/z6kajV0e\n"\
45 "N/t008FdlVKHXAIYWF93JMoVvIpMmT8jft6AN/y3NMpivgt2inmmEJZYNioFJKZG\n"\
46 "X+/vKYvsVISZm2fw8NfnKvAQK55yu+GRWBZGOeS9K+LbYvOwcrjKhHz66m4bedKd\n"\
47 "gVAix6NE5iwmjNXktSQlJMCjbtdNXg/xo1/G4kG2p/MO1HLcKfe1N5FgBiXj3Qjl\n"\
48 "vgvjJZkh1as2KTgaPOBqZaP03738VnYg23ISyvfT/teArVGtxrmFP7939EvJFKpF\n"\
49 "1wTxuDkCgYEA7t0DR37zt+dEJy+5vm7zSmN97VenwQJFWMiulkHGa0yU3lLasxxu\n"\
50 "m0oUtndIjenIvSx6t3Y+agK2F3EPbb0AZ5wZ1p1IXs4vktgeQwSSBdqcM8LZFDvZ\n"\
51 "uPboQnJoRdIkd62XnP5ekIEIBAfOp8v2wFpSfE7nNH2u4CpAXNSF9HsCgYEA2l8D\n"\
52 "JrDE5m9Kkn+J4l+AdGfeBL1igPF3DnuPoV67BpgiaAgI4h25UJzXiDKKoa706S0D\n"\
53 "4XB74zOLX11MaGPMIdhlG+SgeQfNoC5lE4ZWXNyESJH1SVgRGT9nBC2vtL6bxCVV\n"\
54 "WBkTeC5D6c/QXcai6yw6OYyNNdp0uznKURe1xvMCgYBVYYcEjWqMuAvyferFGV+5\n"\
55 "nWqr5gM+yJMFM2bEqupD/HHSLoeiMm2O8KIKvwSeRYzNohKTdZ7FwgZYxr8fGMoG\n"\
56 "PxQ1VK9DxCvZL4tRpVaU5Rmknud9hg9DQG6xIbgIDR+f79sb8QjYWmcFGc1SyWOA\n"\
57 "SkjlykZ2yt4xnqi3BfiD9QKBgGqLgRYXmXp1QoVIBRaWUi55nzHg1XbkWZqPXvz1\n"\
58 "I3uMLv1jLjJlHk3euKqTPmC05HoApKwSHeA0/gOBmg404xyAYJTDcCidTg6hlF96\n"\
59 "ZBja3xApZuxqM62F6dV4FQqzFX0WWhWp5n301N33r0qR6FumMKJzmVJ1TA8tmzEF\n"\
60 "yINRAoGBAJqioYs8rK6eXzA8ywYLjqTLu/yQSLBn/4ta36K8DyCoLNlNxSuox+A5\n"\
```

```
"w6z2vEfRVQDq4Hm4vBzjdi3QfYLNkTiTqLcvgWZ+eX44ogXtdTDO7c+GeMKWz4XX\n"\
"uJSUVL5+CVjKLjZEJ6Qc2WZLl94xSwL71E41H4YciVnSCQxVc4Jw\n"\
"-----END␣RSA␣PRIVATE␣KEY-----\n";


FILE *f;
int padding = RSA_PKCS1_PADDING;

RSA * createRSA(unsigned char * key,int public_internal)
{
    RSA *rsa= NULL;
    BIO *keybio ;
    keybio = BIO_new_mem_buf(key, -1);
    if (keybio==NULL)
    {
        printf( "Failed␣to␣create␣key␣BIO");
        return 0;
    }
    if(public_internal)
    {
        rsa = PEM_read_bio_RSA_PUBKEY(keybio, &rsa,NULL, NULL);
    }
    else
    {
        rsa = PEM_read_bio_RSAPrivateKey(keybio, &rsa,NULL, NULL);
    }
    if(rsa == NULL)
    {
        printf( "Failed␣to␣create␣RSA");
    }

    return rsa;
}
```

```
94
95  int public_encrypt(unsigned char * data,int data_len,unsigned char * key,
        unsigned char *encrypted)
96  {
97      RSA * rsa = createRSA(key,1);
98      int result = RSA_public_encrypt(data_len,data,encrypted,rsa,padding);
99      return result;
100 }
101 int private_decrypt(unsigned char * enc_data,int data_len,unsigned char *
        key, unsigned char *decrypted)
102 {
103     RSA * rsa = createRSA(key,0);
104     int result = RSA_private_decrypt(data_len,enc_data,decrypted,rsa,
            padding);
105     return result;
106 }
107
108
109 int private_encrypt(unsigned char * data,int data_len,unsigned char * key,
         unsigned char *encrypted)
110 {
111     RSA * rsa = createRSA(key,0);
112     int result = RSA_private_encrypt(data_len,data,encrypted,rsa,padding);
113     return result;
114 }
115 int public_decrypt(unsigned char * enc_data,int data_len,unsigned char *
        key, unsigned char *decrypted)
116 {
117     RSA * rsa = createRSA(key,1);
118     int result = RSA_public_decrypt(data_len,enc_data,decrypted,rsa,
            padding);
119     return result;
120 }
```

```
121
122 void printLastError(char *msg)
123 {
124     char * err = (char*)malloc(130);;
125     ERR_load_crypto_strings();
126     ERR_error_string(ERR_get_error(), err);
127     printf("%s␣ERROR:␣%s\n",msg, err);
128     free(err);
129 }
130
131
132 DWORD WINAPI MyThreadFunction0( LPVOID lpParam )
133 {
134   char plainText[2048/8];
135   int id = *((int*)lpParam); //global variable
136   printf("%d,",id);
137
138   sprintf(plainText,"%d",id);
139
140     unsigned char encrypted[4098]={};
141   private_encrypt((unsigned char*)plainText,strlen(plainText),(unsigned
        char*)privateKey,encrypted );
142
143 if (f){
144
145   fwrite(encrypted, 4098, 1, f);
146 }
147 else{
148     puts("Something␣wrong␣writing␣to␣File.\n");
149 }
150
151   return 0;
152 }
```

```
153
154 int main(int argc, char* argv[])
155 {
156
157   if(argc != 3)
158   {
159     printf("Usage: RDTSC <number of threads> <output file>\n");
160   }
161   //f = fopen("test.txt","ab");
162   f = fopen(argv[2],"ab");
163
164   srand(time(0));
165   int threadCount=strtol(argv[1],NULL,10);
166   HANDLE* threads;
167
168   threads = (HANDLE*)malloc(threadCount);
169
170     int i = 0;
171   for(int i=0;i<threadCount;i++)
172   {
173     threads[i]=CreateThread(NULL, 0, MyThreadFunction0, (LPVOID)&i, 0,
          NULL);
174
175   }
176   WaitForMultipleObjects(threadCount, threads, TRUE,INFINITE);
177   printf("\n");
178   free(threads);
179   return 0;
180 }
```

A.2 Decryption Code

```
1 #include <openssl/pem.h>
2 #include <openssl/ssl.h>
```

```
3  #include <openssl/rsa.h>
4  #include <openssl/evp.h>
5  #include <openssl/bio.h>
6  #include <openssl/err.h>
7  #include <stdio.h>
8
9  int padding = RSA_PKCS1_PADDING;
10
11 RSA * createRSA(unsigned char * key,int public_internal)
12 {
13     RSA *rsa= NULL;
14     BIO *keybio ;
15     keybio = BIO_new_mem_buf(key, -1);
16     if (keybio==NULL)
17     {
18         printf( "Failed␣to␣create␣key␣BIO");
19         return 0;
20     }
21     if(public_internal)
22     {
23         rsa = PEM_read_bio_RSA_PUBKEY(keybio, &rsa,NULL, NULL);
24     }
25     else
26     {
27         rsa = PEM_read_bio_RSAPrivateKey(keybio, &rsa,NULL, NULL);
28     }
29     if(rsa == NULL)
30     {
31         printf( "Failed␣to␣create␣RSA");
32     }
33
34     return rsa;
35 }
```

```
36
37 int public_encrypt(unsigned char * data,int data_len,unsigned char * key,
       unsigned char *encrypted)
38 {
39     RSA * rsa = createRSA(key,1);
40     int result = RSA_public_encrypt(data_len,data,encrypted,rsa,padding);
41     return result;
42 }
43 int private_decrypt(unsigned char * enc_data,int data_len,unsigned char *
       key, unsigned char *decrypted)
44 {
45     RSA * rsa = createRSA(key,0);
46     int result = RSA_private_decrypt(data_len,enc_data,decrypted,rsa,
           padding);
47     return result;
48 }
49
50
51 int private_encrypt(unsigned char * data,int data_len,unsigned char * key,
        unsigned char *encrypted)
52 {
53     RSA * rsa = createRSA(key,0);
54     int result = RSA_private_encrypt(data_len,data,encrypted,rsa,padding);
55     return result;
56 }
57 int public_decrypt(unsigned char * enc_data,int data_len,unsigned char *
       key, unsigned char *decrypted)
58 {
59     RSA * rsa = createRSA(key,1);
60     int result = RSA_public_decrypt(data_len,enc_data,decrypted,rsa,
           padding);
61     return result;
62 }
```

```
63
64 void printLastError(char *msg)
65 {
66     char * err = (char*)malloc(130);;
67     ERR_load_crypto_strings();
68     ERR_error_string(ERR_get_error(), err);
69     printf("%s␣ERROR:␣%s\n",msg, err);
70     free(err);
71 }
72
73 int main(int argc, char* argv[]){
74
75   //FILE *f = fopen("file.txt","wb");
76   int threadCount = strtol(argv[1],NULL,10);
77   FILE *f = fopen(argv[2],"rb");
78   char plainText[2048/8] = "Hello␣this␣is␣JB"; //key length : 2048
79
80  char publicKey[]="-----BEGIN␣PUBLIC␣KEY-----\n"\
81 "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAy8Dbv8prpJ/0kKhlGeJY\n"\
82 "ozo2t60EG8L0561g13R29LvMR5hyvGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+\n"\
83 "vw1HocOAZtWK0z3r26uA8kQYOKX9Qt/DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQAp\n"\
84 "fc9jB9nTzphOgM4JiEYvlV8FLhg9yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68\n"\
85 "i6T4nNq7NWC+UNVjQHxNQMQMzU6lWCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoV\n"\
86 "PpY72+eVthKzpMeyHkBn7ciumk5qgLTEJAfWZpe4f4eFZj/Rc8Y8Jj2IS5kVPjUy\n"\
87 "wQIDAQAB\n"\
88 "-----END␣PUBLIC␣KEY-----\n";
89
90  char privateKey[]="-----BEGIN␣RSA␣PRIVATE␣KEY-----\n"\
91 "MIIEowIBAAKCAQEAy8Dbv8prpJ/0kKhlGeJYozo2t60EG8L0561g13R29LvMR5hy\n"\
92 "vGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+vw1HocOAZtWK0z3r26uA8kQYOKX9\n"\
93 "Qt/DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQApfc9jB9nTzphOgM4JiEYvlV8FLhg9\n"\
94 "yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68i6T4nNq7NWC+UNVjQHxNQMQMzU6l\n"\
95 "WCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoVPpY72+eVthKzpMeyHkBn7ciumk5q\n"\
```

```
96  "gLTEJAfWZpe4f4eFZj/Rc8Y8Jj2IS5kVPjUywQIDAQABAoIBADhg1u1Mv1hAAlX8\n"\
97  "omz1Gn2f4AAW2aos2cM5UDCNw1SYmj+9SRIkaxjRsE/C4o9sw1oxrg1/z6kajV0e\n"\
98  "N/t008FdlVKHXAIYWF93JMoVvIpMmT8jft6AN/y3NMpivgt2inmmEJZYNioFJKZG\n"\
99  "X+/vKYvsVISZm2fw8NfnKvAQK55yu+GRWBZGOeS9K+LbYvOwcrjKhHz66m4bedKd\n"\
100 "gVAix6NE5iwmjNXktSQlJMCjbtdNXg/xo1/G4kG2p/MO1HLcKfe1N5FgBiXj3Qjl\n"\
101 "vgvjJZkh1as2KTgaPOBqZaP03738VnYg23ISyvfT/teArVGtxrmFP7939EvJFKpF\n"\
102 "1wTxuDkCgYEA7t0DR37zt+dEJy+5vm7zSmN97VenwQJFWMiulkHGa0yU3lLasxxu\n"\
103 "m0oUtndIjenIvSx6t3Y+agK2F3EPbb0AZ5wZ1p1IXs4vktgeQwSSBdqcM8LZFDvZ\n"\
104 "uPboQnJoRdIkd62XnP5ekIEIBAfOp8v2wFpSfE7nNH2u4CpAXNSF9HsCgYEA2l8D\n"\
105 "JrDE5m9Kkn+J4l+AdGfeBL1igPF3DnuPoV67BpgiaAgI4h25UJzXiDKKoa706S0D\n"\
106 "4XB74zOLX11MaGPMIdhlG+SgeQfNoC5lE4ZWXNyESJH1SVgRGT9nBC2vtL6bxCVV\n"\
107 "WBkTeC5D6c/QXcai6yw6OYyNNdp0uznKURe1xvMCgYBVYYcEjWqMuAvyferFGV+5\n"\
108 "nWqr5gM+yJMFM2bEqupD/HHSLoeiMm2O8KIKvwSeRYzNohKTdZ7FwgZYxr8fGMoG\n"\
109 "PxQ1VK9DxCvZL4tRpVaU5Rmknud9hg9DQG6xIbgIDR+f79sb8QjYWmcFGc1SyWOA\n"\
110 "SkjlykZ2yt4xnqi3BfiD9QKBgGqLgRYXmXp1QoVIBRaWUi55nzHg1XbkWZqPXvz1\n"\
111 "I3uMLv1jLjJlHk3euKqTPmCO5HoApKwSHeA0/gOBmg404xyAYJTDcCidTg6hlF96\n"\
112 "ZBja3xApZuxqM62F6dV4FQqzFX0WWhWp5n301N33r0qR6FumMKJzmVJ1TA8tmzEF\n"\
113 "yINRAoGBAJqioYs8rK6eXzA8ywYLjqTLu/yQSLBn/4ta36K8DyCoLNlNxSuox+A5\n"\
114 "w6z2vEfRVQDq4Hm4vBzjdi3QfYLNkTiTqLcvgWZ+eX44ogXtdTDO7c+GeMKWz4XX\n"\
115 "uJSUVL5+CVjKLjZEJ6Qc2WZLl94xSwL71E41H4YciVnSCQxVc4Jw\n"\
116 "-----END␣RSA␣PRIVATE␣KEY-----\n";
117
118
119 unsigned char encrypted[4098]={};
120 unsigned char decrypted[4098]={};
121
122 int encrypted_length,decrypted_length;
123
124 //const int threadCount=25;
125 for(int i = 0;i<threadCount-1;i++)
126 {
127
128   encrypted_length=256;
```

```
129
130   if (f){
131     fread(encrypted,4098, 1, f);
132     //puts("Read from file!");
133
134   }
135   else{
136     puts("Something␣wrong␣reading␣from␣File.\n");
137   }
138
139   encrypted_length=256;
140
141   decrypted_length = public_decrypt(encrypted,encrypted_length,(unsigned
          char*)publicKey, decrypted);
142   if(decrypted_length == -1)
143   {
144     printLastError("Public␣Decrypt␣failed");
145     exit(0);
146   }
147   printf("%s,",decrypted);
148   //printf("Decrypted Length =%d\n",decrypted_length);
149 }
150
151   decrypted_length = public_decrypt(encrypted,encrypted_length,(unsigned
          char*)publicKey, decrypted);
152   printf("%s\n",decrypted);
153
154
155 }
```