SANDIA REPORT

SAND2013-2986

Unlimited Release

Printed March 2013

A Flexible Framework for Secure and Efficient Program Obfuscation

John H. Solis

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

A Flexible Framework for Secure and Efficient Program Obfuscation

John H. Solis
Sandia National Laboratories
Livermore, CA, USA
jhsolis@sandia.gov

Abstract

In this paper, we present a modular framework for constructing a secure and efficient program obfuscation scheme. Our approach, inspired by the *obfuscation with respect to oracle machines* model of [4], retains an interactive online protocol with an oracle, but relaxes the original computational and storage restrictions. We argue this is reasonable given the computational resources of modern personal devices. Furthermore, we relax the information-theoretic security requirement for computational security to utilize established cryptographic primitives.

With this additional flexibility we are free to explore different cryptographic building-blocks. Our approach combines authenticated encryption with private information retrieval to construct a secure program obfuscation framework. We give a formal specification of our framework, based on desired functionality and security properties, and provide an example instantiation. In particular, we implement AES in Galois/Counter Mode for authenticated encryption and the Gentry-Ramzan [13] constant communication-rate private information retrieval scheme. We present our implementation results and show that non-trivial sized programs can be realized, but scalability is quickly limited by computational overhead. Finally, we include a discussion on security considerations when instantiating specific modules.

Acknowledgments

This work was funded by the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

1	Introduction and Motivation	7
2	Related Work	9
3	Preliminaries	11
3.1	CodeSeal Overview	11
3.2	Proposed Approach	12
4	Secure Program Obfuscation Framework	13
4.1	Primary Components	13
4.2	Operation Phases	13
4.3	Compiler	14
4.4	Client	15
4.5	Oracle	16
5	Formal Specification	17
5.1	Model and Definition	17
6	Implementation Details	19
6.1	Benchmarks	21
6.2	Security Considerations	22
7	Conclusions	25
	References	26

1 Introduction and Motivation

Program obfuscation is a software protection technique that attempts to hide internal instructions or operations from unauthorized observers. In practice, this is accomplished by transformation tools that convert a given piece of source code into a new program that is functionally equivalent, yet completely unintelligible from the original. Several commercial and open-source obfuscators exist for a wide variety of languages, including, C/C++ [24], Java [3, 23], and JavaScript [17].

However, these obfuscators are far from ideal because they cannot prevent a dedicated adversary from inverting the transformations that create the obfuscated code. Such reverse-engineering is possible because obfuscated code must be compiled into a form that is eventually executed by another computer, i.e., x86 instructions or Java byte-code. Since the compiled form retains all the information of the un-obfuscated code, the original functionality can be recovered given the appropriate decompiler, time and patience.

An ideal obfuscator, on the other hand, would transform a program into a “virtual black box” that, when executed, leaks no information about the underlying program instructions. Unfortunately, general program obfuscation in offline settings has been shown to be theoretically impossible [5, 15]. Despite these results, alternative obfuscation models have been proposed to investigate scenarios where secure program obfuscation is still possible.

One model, proposed by researchers at Sandia National Laboratories, *obfuscation with respect to oracle machines* [4] led to the development of CodeSeal [8] – a provably secure code obfuscation technology. Unfortunately, due to certain assumptions in the original model, it is primarily limited to small and trivial programs. Extending its capabilities to standard architectures, e.g. x86 and ARM, as well as, improving its performance characteristics, would enable larger and more complex programs to run securely in untrusted environments while simultaneously preserving program integrity from malicious tampering.

Our goal is to build upon this effort and extend the practical applicability of the scheme to standard architectures while preserving the desired security properties. To achieve this, we retain the interactive online protocol with an oracle, but relax the computational and storage restrictions. We argue this is reasonable given the computational resources of modern personal devices. Even constrained microprocessors, e.g., smartcards, contain several hundred kilobytes of memory. Furthermore, we relax the original information-theoretic security requirement for computational security to utilize advanced cryptographic primitives. With this added flexibility, we explore composing different cryptographic building-blocks into a single secure program obfuscation framework.

Contributions: In this work, we develop a modular framework for constructing secure and efficient program obfuscation schemes. We give a formal specification of our framework, based on desired functionality, integrity and security properties, and focus on combining authenticated encryption with private information retrieval (PIR). In particular, we implement AES in Galois/Counter Mode for authenticated encryption and the Gentry-Ramzan [13] constant communication-rate PIR scheme. Our results show that non-trivial sized programs can be realized for standard architectures. Finally, we discuss security considerations when instantiating specific modules.

2 Related Work

Although not always identified explicitly, program obfuscation has been studied in the context of several closely related problems:

Software Protection Problem: Protecting software from unauthorized tampering and/or duplication is an open problem that costs the software industry billions of dollars annually in lost revenue. The earliest anti-piracy and software protection work dates to 1980 by Kent [18], who suggested the use of tamper-resistant trusted-hardware and encrypted programs¹ and differentiated between the trusted host and trusted code problems. A few years later, Gosler [16] proposed the use of dongles and magnetic signatures in floppy drives combined with anti-debugging techniques to prevent software analysis and copying.

Cohen [11] proposed software diversity and code obfuscation, based on simple code transformation and obfuscation techniques, as a software protection mechanism. Additional techniques were later proposed by Collberg *et al* [12] and Wang [25]. These, and other related transformations, can be found in commercial and open-source obfuscators [3, 17, 23, 24].

The first formal treatment of program obfuscation was initiated by Barak *et al* [5] who define an obfuscator O to be an efficient, probabilistic compiler that transforms a program P into functionally equivalent obfuscated program $O(P)$ that behaves like a “virtual black-box”. In essence, the information extracted by $O(P)$ is equivalent to the information that can be extracted given only oracle access to P .

Unfortunately, the primary contribution of this work is a negative result that proves the existence of a family of programs that are non-learnable and yet, unobfuscatable (by any code obfuscator). This implies, in its most extreme interpretation, that there does not exist a provably secure obfuscation algorithm that works on every program. The final conclusion is that the offline “virtual black box” concept is fundamentally flawed and that different models of program obfuscation must be considered.

In one different model, *obfuscation with respect to oracle machines* [4], the authors propose a provably secure program obfuscation technology applicable to deterministic finite automata and general Turing machines. The obfuscator takes as input an unobfuscated program and outputs obfuscated code along with its corresponding oracle. Our work extends this basic model and proposes techniques to improve its performance. We provide an extended overview in Section 3.1.

Private Information Retrieval and Oblivious RAM: Although not directly related to obfuscation, private information retrieval (PIR) and oblivious RAM (ORAM) schemes are worth mentioning since this is the primary mechanism by which we seek to improve the communication protocol. ORAM [14] schemes hide data access patterns in conjunction with trusted hardware to prevent illegal software replication. Private information retrieval is a cryptographic mechanism in which a client can query a database server for a record at index i , without revealing i to the server. Several schemes exist to ensure private queries in both multi-server [9, 10] and single-server set-

¹At the highest level of abstraction, a similar approach is taken in this work

tings [7, 13, 20, 26]. It has also been shown that PIR schemes are closely connected to other cryptographic primitives [22].

In our framework, we base the client-to-oracle communication protocol on the Gentry-Ramzan [13] scheme. This constant communication-rate PIR scheme for the single-server setting is ideal for minimizing communication complexity. However, this approach, based on a variant of the ϕ -hiding assumption, requires a particularly expensive operation: computing discrete logarithms on smooth order subgroups. Although asymptotically efficient, larger key sizes will be the limiting factor in determining efficient query sizes in practice. As part of our work, we analyze this practical trade-off between the computational and communication complexity as applied to program obfuscation.

Finally, we note there have been several PIR/ORAM implementation studies as applied to a variety of domains. These include a mechanism for private DNS queries [21], outsourced databases [26], and location privacy for location-based services [19].

3 Preliminaries

3.1 CodeSeal Overview

CodeSeal [8] is a provably secure code obfuscation technology developed at Sandia National Laboratories that operates under a new model called “obfuscation with respect to oracle machines”. This approach, applicable to both deterministic finite automata (DFA) and general Turing machines (TM), takes as input an unobfuscated program and outputs: (1) obfuscated code functionally equivalent to the original, and (2) an oracle capable of executing the obfuscated code. We now give a brief overview of the work and point the reader to [4] for complete details.

Under the oracle machine model, obfuscation is modeled as an online two-party protocol, where the first party (client) stores some obfuscated code and the second party (oracle) contains a small secret. The authors show that the oracle’s internal state must be read-proof and non-static as a necessary condition for secure obfuscation, i.e., a trusted or tamper-resistant oracle. Otherwise, an adversary could simply clone or virtualize the oracle.

Additionally, the oracle’s storage capacity was restricted to being asymptotically smaller than the unobfuscated program. This critical restriction was imposed to prevent the trivial solution where the oracle devolves into a remote-procedure-call (RPC) server storing all programs a-priori. In practice, this models a computationally limited device, such as, a smartcard or co-processor.

The obfuscator, O , is a compiler that converts a program, P , into obfuscated code, $O(P)$, that is functionally equivalent to the original. P is represented as a TM state transition table and $O(P)$ is the authenticated encryption of P . Authenticated encryption preserves both the integrity and confidentiality of P .

The obfuscator also outputs an oracle that internally contains the corresponding decryption key. To execute obfuscated code, the client sends $O(P)$ to the oracle who then decrypts and executes the state-transition function based on the current input. Because of the imposed storage constraints, the obfuscated code (and encrypted working tape information) must be retransmitted for every state-transition. The final accept/reject decision is (optionally) returned to client.

Since the obfuscated code is encrypted, it can be distributed to any number of (untrusted) users or devices. No other information is leaked to the client since the encrypted obfuscated code is indistinguishable from random without knowledge of the decryption key. This holds even if the user device has been compromised by a malicious adversary.

Although this is a very powerful primitive, communication overhead limits the approach to small and trivial programs. This is a result of the storage constraints and information-theoretic security requirement, and require that the entire obfuscated state transition table and working tape be transmitted for each individual state transition. Additionally, the communication overhead cannot easily be bounded since the number of instructions executed is often much greater than the actual program size, e.g., due to repeated instructions (for and while loops).

This final point is acknowledged in the original analysis, with the authors noting that the oracle’s internal variables may grow exponentially large and overflow the assumed storage space. To prevent this end-state, the authors propose releasing the decryption key and allowing the obfuscated code to be decrypted and run in the open. However, this is not an ideal solution.

3.2 Proposed Approach

We argue that the computational and storage restrictions originally placed on the oracle are too restrictive and unrealistic considering the computational abilities of modern devices. Even the most constrained microprocessors found in smartcards contain several hundred kilobytes of flash memory. We propose relaxing these assumptions and, additionally, relaxing the original information-theoretic security requirement for computational security to utilize advanced cryptographic primitives. The flexibility we gain by doing so, opens up the possibility of exploring different approaches to secure program obfuscation.

We use this freedom to construct a secure program-obfuscation framework based on the composition of authenticated encryption and private information retrieval primitives. Our hypothesis is that replacing the original unbounded communication protocol with the Gentry-Ramzan [13] constant communication-rate private information retrieval scheme will allow larger programs to be obfuscated. Intuitively, this holds because a PIR scheme removes the need to transfer the entire state transition table to execute a single state-transition operation. However, the computational overhead required by PIR will eventually limit the performance improvements gained in practice.

4 Secure Program Obfuscation Framework

We now give a high level overview of our secure program obfuscation framework:

4.1 Primary Components

Figure 1: Primary Framework Components

Figure 1 illustrates the primary components of our secure program obfuscation framework:

Compiler: The compiler is responsible for transforming an unobfuscated program into a functionally equivalent obfuscated form.

Client: Clients store a set of obfuscated programs that are executed via an interactive protocol with their respective oracle. Clients are capable of providing program inputs and (optionally) learn output behavior. However, they remain completely oblivious to the exact functionality of the obfuscated program.

Oracle: The oracle is the most complex of the primary components. It is responsible for de-obfuscating and executing programs in a trusted local environment. Once program execution has terminated, the oracle (optionally) returns any program output.

4.2 Operation Phases

Component operations are divided into *compile time* and *runtime* phases as depicted in Figure 2. The compiler is only utilized during the compile-time phase and, whenever possible, in a secure or trusted environment due to generation of sensitive cryptographic keys used during the obfuscation process. The *Target* input refers to the target CPU architecture, i.e., x86 or ARM.

Compile Time

Run Time

Figure 2: Operation phases

During the runtime phase, the client and oracle execute an obfuscated program, $O(P)$, via a variable-round interactive protocol. The exact number of rounds is determined by the oracle and its local storage resources, as well as, the program P . It is important to note that all protocol messages (apart from the client input) are completely obfuscated, i.e., only valid oracles are capable of decrypting messages. However, the oracle may return the final program output message as plaintext if required for a particular scenario.

4.3 Compiler

Figure 3: Compiler

The compiler is responsible for transforming programs into a cryptographically secure obfuscated form. It is divided into the series of independent sub-components as shown in Figure 3. Conceptually, the compiler should be capable of accepting the program, target instruction set, and

cryptographic secret key as inputs, and output the obfuscated program $O(P)$ and oracle capable of executing the obfuscated program.

In our x86/ARM implementation, the Compile-To-Target component is simply the traditional GNU Toolchain. It accepts x86 and ARM assembly source code files and compiles the program into the appropriate instruction set. After compilation, the compiled binary is encrypted using secret key input with one of the following encryption methods: No Encryption (debugging purposes) or AES in GCM mode. This encrypted binary is the obfuscated program output by the compiler. The oracle, as we will see in Section 4.5, is a general (prototype) oracle that is not tied to a specific obfuscated program. However, it is tied to a specific instruction set, i.e., an x86 oracle cannot execute on an ARM processor.

4.4 Client

Figure 4: Client operation and pseudo-code

The client is the simplest primary component. It is composed of internal storage and a communication sub-components. We assume that clients have the internal storage capacity to store the complete obfuscated program $O(P)$. A client may contain multiple obfuscated programs and may communicate with one (or more) oracles during runtime. Although not specified here, we assume that a client knows the correct mapping from $O(P)$ to runtime oracle.

In our x86/ARM implementation, the communication component can be instantiated as either a TCP/IP socket or as a Serial-over-USB link. The exact instantiation is compiled directly into the client and may vary depending on the use case. For example, an oracle with access to the computer network would use the TCP/IP socket interface while a compact Gumstix system-on-chip device would utilize a serial-over-USB communication link.

4.5 Oracle

Figure 5: Oracle operation and pseudo-code

The oracle is the most complex of the primary components with four independent sub-components. The oracle receives an obfuscated program and (optional) client input via the communication component. As with the client communication component, this can be either a TCP/IP socket or Serial-over-USB link. It is important to note that the components are not interchangeable, i.e., an instance of the socket communication component can only communicate with other socket communication components.

After the input has been received, the oracle de-obfuscates the obfuscated program by decrypting it with the appropriate decryption method and decryption key. If the decryption component fails, e.g., an invalid decryption method selected, the oracle aborts the runtime protocol and notifies the client of the error. Otherwise, the oracle feeds the unobfuscated program into the runtime engine component. This component is responsible for executing the next instruction(s) and (optionally) returning updated registers and memory contents.

5 Formal Specification

The previous section outlined a high-level overview of our approach and described several interacting components. This helps understand the additional complexity involved when implementing our scheme for modern architectures.

However, our formal specification abstracts away from these details and focuses on two key cryptographic components: authenticated encryption and private information retrieval. These cryptographic primitives contain the security and privacy properties that, when composed together, satisfy the requirements of a secure program obfuscation scheme.

Informally, we want an obfuscator that converts programs into virtual black boxes and a two-party execution protocol that is both secure and private. This prevents a malicious adversary from learning access patterns and from manipulating the obfuscated program.

5.1 Model and Definition

We expand the definition of polylogarithmic private information retrieval from [13] to include security and integrity properties for the database contents. Security is defined as the advantage an adversary has in the indistinguishability under chosen-ciphertext attack game (*IND-CCA*). A similar definition is used for the integrity of ciphertexts (*INT-CTXT*) game. We adopt the formal definitions of *IND-CCA* and *INT-CTXT* from [6].

Notation: If S is a set of elements, and D is a sampleable probability distribution on S , then let $s \xleftarrow{D} S$ denote the process of picking an element s from S according to distribution D .

Let $A(\cdot, \dots, \cdot)$ denote a probabilistic polynomial time algorithm that takes one or more inputs. Let $\Pr[y \leftarrow A(x) : b(y)]$ denote the probability that $b(y)$ is true after y was generated by A on input x . We denote $A_Q^B(\cdot)$ as an algorithm that makes queries to B using algorithm Q .

We denote $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ as a standard symmetric encryption scheme. $\mathcal{K}$ is the key generation algorithm; $\mathcal{E}$ is the encryption algorithm that takes as inputs a key and plaintext message and outputs a ciphertext; $\mathcal{D}$ is the decryption algorithm that takes as inputs a key and ciphertext and outputs the corresponding plaintext.

We call algorithm $O(\mathcal{SE}, P)$ an obfuscator that takes as inputs a symmetric encryption scheme and program P and outputs the ciphertext ObP , i.e., execute the encryption algorithm of $\mathcal{SE}$. Let n' and n denote the size of P and ObP in bits, respectively. For $a, b \in \mathbb{Z}$ with $a \leq b$, let $[a, b]$ denote the set of integers between a and b inclusive and $[b]$ denote $[1, b]$. Finally, let k and k' be security parameters of the system. We explain the relationship between k and k' below.

Definition 1. Let $Q(\cdot, \cdot, \cdot)$, $D(\cdot, \cdot, \cdot)$, $O(\cdot, \cdot)$, $R(\cdot, \cdot, \cdot)$, and be polynomial-time algorithms. We say that (Q, D, O, R) is a fully secure program obfuscation scheme if there exist constants $a, b, c, d > 0$ such that:

- (Efficient Obfuscator) $\forall n' \in \mathbb{N}$, there exists a polynomial p such that,

$$|O(\mathcal{SE}, P)| \leq p(n') \leq n$$

- (Efficient Query Protocol) $\forall n' \in \mathbb{N}$, and if P takes t steps on input x , then there exists a polynomial p such that $A_Q^R(x)$ takes at most $p(t+n')$ steps on x
- (Query Correctness) $\forall n \in \mathbb{N}$, $\forall B \in \{0, 1\}^n$, $\forall i \in [1, n]$, and $\forall k' \in \mathbb{N}$,

$$\Pr[(q, s) \xleftarrow{R} Q(n, i, 1^{k'}); r \xleftarrow{R} D(B, q, 1^{k'}) : R(n, i, (q, s), r, 1^{k'}) = B_i] > 1 - 2^{-ak'}$$

- (Query Privacy) $\forall n \in \mathbb{N}$, $\forall i, j \in [1, n]$, $\forall k' \in \mathbb{N}$ such that $2^{k'} > n^b$, and $\forall 2^{ck'} - \text{gate circuits } A$,
- $$|\Pr[(q, s) \xleftarrow{R} Q(n, i, 1^{k'}) : A(n, q, 1^{k'}) = 1] - \Pr[(q, s) \xleftarrow{R} Q(n, j, 1^{k'}) : A(n, q, 1^{k'}) = 1]| < 2^{-dk'}$$
- (Obfuscated Program Integrity) We say adversary A wins in the INT-CTXT game if it submits to a verifier a ciphertext C not previously returned by $\mathcal{E}$:

$$\text{Adv}^{\text{int-ctxt}}(A) = \Pr[\text{INT} - \text{CTXT}^A \implies 1]$$

- (Obfuscated Program Security) We say adversary A wins in the IND-CCA game if it

$$\text{Adv}^{\text{ind-cca}}(A) = 2 \times \Pr[\text{IND} - \text{CCA}^A \implies 1] - 1$$

- (Virtual Black Box) $\forall k' \in \mathbb{N}$, for every algorithm A and input x , there is a simulator S , such that,

$$|\Pr[A_Q^R(O_Q^R(\mathcal{SE}, P), 1^k, x) = 1] - \Pr[S_Q^M(1^{|P|}, 1^k, x) = 1]| < 2^{-k}$$

Here a, b, c, d are the fundamental constants of the underlying CPIR scheme. Let $B \leftarrow O(\mathcal{SE}, P)$ be the obfuscated program output that become the contents of our database, D is the client's response algorithm; Q is the oracle's query-generating algorithm; R is the oracle's recovery algorithm; q is the oracle's query to the client; s is the oracle's secret (associated with q); e is the client's response. Note that the client/oracle are inverted from the usual user and database roles in PIR. In our scheme, the oracle queries a client storing the obfuscated program.

Our scheme requires two separate security parameters: k and k' . The k parameter is for the symmetric authenticated encryption scheme $\mathcal{SE}$ and k' for the PIR scheme. The two are related by $k' = f(k)$ for some polynomial f . In practice, we may have $k' = \max\{1024, |\mathcal{E}| * 4\}$ to ensure that the blocksize retrieved during each query round is large enough to contain at least a single ciphertext.

6 Implementation Details

We implemented our scheme in C++ with multiple open source libraries: our Gentry-Ramzan PIR scheme is based on the GNU Multiple Precision Arithmetic Library [1] and use the block-cipher implementations found in the OpenSSL library [2]. Figure 6 depicts the inheritance diagram of all possible engine instantiations. All three of the primary components (compiler, oracle, and client) are built from the same class implementations.

Figure 6: Engine inheritance diagram

Makefile Variables

The specific engines are selected at compile-time by setting the appropriate Makefile variables in the top-level directory.

- **COMMUNICATION**=[**tcp** | **serialusb** | **all**]

This variable defines the communication interface used between the oracle and server. The **tcp** option creates an IPv4 TCP/IP sockets communication channel. The **serialusb** option creates a serial-over-USB connection interface (intended for use with system-on-chip Gumstix devices). The **all** option compiles support for both interfaces directly into the oracle.

- **COMM_PROTO**=[**basic** | **cpir**]

This variable defines the communication protocol used between the oracle and server. The **basic** option defines a simple communication protocol where the server transmits all obfuscated code during each communication round. The **cpir** option defines an optimized communication protocol built from the Gentry-Ramzan Constant Private Information Retrieval protocol that allows an oracle to privately query only the data that is needed.

- **COMMPROTO_KEYFILE**=[*key file path*]

This variable defines the secret key file used by the underlying communication protocol. It is only required when **COMM_PROTO**=**cpir** and should contain all the public key material needed to execute the CPIR protocol.

Note: CPIR protocol parameters are specified inside the keyfile. They can be set using the **CMPIR_CPIR_BLOCKSIZE** and **CMPIR_CPIR_KEYSIZE** variables or by executing the provided key generation tool manually.

- **CRYPTO_ENGINE**=[**aes** | **des** | **blowfish** | **rc4** | **none**]

This variable defines the cryptographic engine used to obfuscate and de-obfuscated programs. Currently, the only supported options are **aes**, **des**, **blowfish**, **rc4**, and **none**. The first option defines the AES encryption algorithm in GCM mode and is currently the only algorithm with GCM support in OpenSSL. The second option defines 3-DES encryption in outer triple CBC DES encryption mode (the mode used by SSL). The third option defines the Blowfish encryption algorithm in CBC mode. The fourth option defines the RC4 stream cipher algorithm. The final option performs no program obfuscation and is primarily intended for debugging purposes.

- **CRYPTO_KEYFILE**=[*key file path*]

This variable defines the secret key file used by the underlying cryptographic engine. The key file contains all the key material needed to properly encrypt and decrypt obfuscated code.

- **RUNTIME_ENGINE**=[**gdb** | **native**]

This variable defines the oracle runtime engine used to parse and execute de-obfuscated code. The first option defines a GDB based oracle capable of executing x86 and ARM based instruction sets. The second option defines a “native” engine that executes code as if it were initiated as remote-procedure-call protocol.

Note: The GDB based oracle requires more resources than what is optimally required. The current implementation is a research prototype intended to serve as a proof-of-concept.

- **EXEC_INST_CNT**=[*Numeric value*]

This variable defines the number of instructions that should be executed by the oracle during each communication round. This value will depend on the resources of the local environment available to the oracle and must be greater than zero. In practice, this value should be as large as possible to improve runtime performance.

- **TARGET_SET**=[**x86** | **arm**]

This variable defines the target instruction set for the server and oracle during compilation. The first option defines an Intel x86 instruction set and traditional GNU toolchain. The second defines ARM based instruction set and the *arm-linux-gnueabi* cross-compilation toolchain.

- **ORACLE_INST_SET**=[**x86** | **arm**]

This variable defines the target instruction set that the oracle is expected to parse and execute with the runtime engine.

Note: It is entirely possible for a client to communicate (and execute obfuscated code) with an oracle based on a different instruction set. For example, an x86 based client can communicate with an ARM based Gumstix oracle if the obfuscated code is also ARM based.

Additional documentation for multiple demonstrations, as well as, developer APIs are available upon request as supplementary materials.

6.1 Benchmarks

All tests were performed on a 3.2 GHz Intel Core i7 CPU running the Ubuntu 12.04 Linux distribution. Both the client and server algorithms were benchmarked to gain an understanding of practical performance issues.

The benchmarking algorithm pseudo-code is outlined in Algorithm 1. It is a simple program that takes as input an array of 42-character strings and a test string. If the test string is a member of the array, the algorithm accepts the input, otherwise it is rejected. For the actual benchmarking, the input array is compiled directly into the obfuscated program so that only the testing string is communicated to the oracle each round. We repeat our experiment for varying sizes of Arr as listed in Figure 7.

Input: array Arr of strings
Input: string str
Output: Accept if $str \in Arr$, otherwise Reject

```

result  $\leftarrow$  Reject
foreach string  $s$  in  $Arr$  do
    if  $s$  equals  $str$  then
        result  $\leftarrow$  Accept
    end
end
return result

```

$ Arr $	$ O(P) $ (bytes)	Instructions Executed
10	672	599
20	1090	1099
30	1510	1639
40	1930	2179
50	2350	2719
60	2770	3259
70	3190	3799
80	3610	4339
90	4030	4879
100	4450	5419

Algorithm 1: Benchmarking algorithm

Figure 7: Arr sizes benchmarked

Our results, displayed in Figure 8, shows how, even for relatively small programs, the CPIR approach quickly outperforms the basic CodeSeal approach. If we focus on just the CPIR approaches (Figure 9), we see only a minor difference between using larger key sizes and transmission overhead. This is easily explained by the fact that larger key sizes allow larger blocks to be retrieved during each round. If twice the number of instructions are retrieved per round, then we only query half as often.

However, the stark difference in running times between key sizes indicates the huge trade-off made between computational and communication complexity. Looking at the total running time for each benchmark, Table 1, and average execution time per instruction, Table 2, we see the performance impact of computing discrete logarithms on larger subgroups. This is the clear bottleneck of our scheme and, unfortunately, we quickly hit the limits of (practical) scalability at a 2048 bit key size.

We can conclude, from a practical standpoint, that larger key sizes hinder the overall efficiency of the scheme. Even if program blocks were optimized for locality, e.g., loop operations placed together with inputs, the computational overhead would still be the limiting factor in the long run.

Figure 8: Transmission Overhead

6.2 Security Considerations

There are several security considerations that must be taken into account when implementing a secure program obfuscation scheme using our framework. Perhaps the biggest potential security issue stems from possible side-channel analysis and timing attacks. The two most vulnerable operations are the discrete log computation in the smooth-order subgroup, and the execution of instruction blocks.

The former is vulnerable because the discrete log operation will be computed faster when the database value is small compared to when the value is large. This difference could allow an attacker to identify which block, or subset of blocks, was likely to be queried. Clearly, this violates our *query privacy* requirement. This issue can be avoided by performing a fixed number of operations during each discrete log operation, e.g., always execute the number of operations required for computing the largest block value. Although this approach increases running time, smaller key sizes could help offset the additional overhead.

The second vulnerability is related to the decrypted program instructions. As before, timing differences between certain instructions in modern computing architectures, e.g., floating-point operations vs integer multiplication, may leak information about the original program P . This issue can be addressed by always waiting a fixed number of clock cycles before continuing with normal operation. However, in practice, attacks against this vulnerability would have a low success rate. The overhead from the network communication, decryption, and discrete log operations compensate for the subtle differences during actual program execution.

Figure 9: CPIR Overhead

$ Arr $	Execution Time (min)	
	CPIR-1024	CPIR-2048
10	8.97	130.78
20	16.13	203.58
30	22.35	275.17
40	44.62	571.67
50	51.18	407.10
60	61.37	665.35
70	67.37	491.97
80	53.78	662.70
90	60.33	662.98
100	68.68	780.73

Table 1: Total Execution Time

Scheme	blocksize / keysize (bits)	Per Inst (ms)
CPIR	256 / 1024	945 [182]
CPIR	512 / 2048	10593 [8644]

Table 2: Average Execution Time (std dev in [])

7 Conclusions

We presented a modular framework for constructing a secure and efficient program obfuscation scheme based on the composition of two cryptographic building blocks in a straightforward way that preserved all desired properties. We described a formal specification of our framework, and implemented a single example using AES in Galois/Counter Mode for authenticated encryption and the Gentry-Ramzan constant communication-rate private information retrieval scheme. We presented our implementation results and showed that non-trivial sized programs can be realized, but are quickly limited by the keysize of the PIR scheme. This indicates that the discrete logarithm operation is simply too expensive and limits scalability to larger programs.

References

- [1] The GNU MP Bignum Library. The GNU Multiple Precision Library, <http://www.gmplib.org>.
- [2] OpenSSL: The Open Source toolkit for SSL/TLS. <http://www.openssl.org>.
- [3] Allatori Java Obfuscator - Professional Java Obfuscation. <http://www.allatori.com>.
- [4] Erik W Anderson. On the secure obfuscation of deterministic finite automata. *Cryptology ePrint Archive, Report 2008/184*, Apr 2008.
- [5] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, 2001. Springer-Verlag.
- [6] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, September 2008.
- [7] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'99, pages 402–414, Berlin, Heidelberg, 1999. Springer-Verlag.
- [8] Adrian R Chavez, Erik J Lee, Michael J McDonald, Jay D Patel, Guylaine M Pollock, John H Solis, and Cassandra M Trevino. Protecting SCADA Supply Chains Using Trust Anchors. Technical Report (SAND2012-7683), Sept 2012.
- [9] Benny Chor and Niv Gilboa. Computationally private information retrieval (extended abstract). In *Journal of the ACM*, pages 41–50, 1997.
- [10] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.
- [11] Frederick B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12:565–584, October 1993.
- [12] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.
- [13] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, pages 803–815. Springer-Verlag, 2005.
- [14] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473, May 1996.

- [15] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, pages 553–562, Washington, DC, USA, 2005. IEEE Computer Society.
- [16] James R Gosler. Software protection: myth or reality? In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 140–157, New York, NY, USA, 1986. Springer-Verlag New York, Inc.
- [17] JavaScript Obfuscator - Protect your JavaScript Code. <http://www.jasob.com>.
- [18] Stephen Thomas Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, 1980.
- [19] Ali Khoshgozaran and Cyrus Shahabi. Privacy in location-based applications. chapter Private Information Retrieval Techniques for Enabling Location Privacy in Location-Based Services, pages 59–83. Springer-Verlag, Berlin, Heidelberg, 2009.
- [20] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of the 8th international conference on Information Security*, ISC'05, pages 314–328, Berlin, Heidelberg, 2005. Springer-Verlag.
- [21] Yanbin Lu and Gene Tsudik. Towards plugging privacy leaks in the domain name system. In *Peer-to-Peer Computing 2010*, pages 1–10, 2010.
- [22] Rafail Ostrovsky and William E. Skeith, III. A survey of single-database private information retrieval: techniques and applications. In *Proceedings of the 10th international conference on Practice and theory in public-key cryptography*, PKC'07, pages 393–411, Berlin, Heidelberg, 2007. Springer-Verlag.
- [23] ProGuard. <http://proguard.sourceforge.net>.
- [24] Stunnix. C++ Obfuscator - obfuscate C and C++ code. <http://www.stunnix.com>.
- [25] Chenxi Wang. *A security architecture for survivability mechanisms*. PhD thesis, Charlottesville, VA, USA, 2001. AAI3000162.
- [26] Peter Williams and Radu Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

DISTRIBUTION:

1	MS 9158	John H. Solis, 8961
1	MS 9158	Keith Vanderveen, 8961
1	MS 0899	Technical Library, 8944 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911