Cryptographic Software IP Protection without Compromising Performance or Timing Side-channel Leakage

ARNAB KUMAR BISWAS, Electrical and Computer Engineering department, National University of Singapore

Program obfuscation is a widely used cryptographic software intellectual property (IP) protection technique against reverse engineering attacks in embedded systems. However, very few works have studied the impact of combining various obfuscation techniques on the obscurity (difficulty of reverse engineering) and performance (execution time) of obfuscated programs. In this article, we propose a Genetic Algorithm (GA)-based framework that not only optimizes obscurity and performance of obfuscated cryptographic programs, but it also ensures very low timing side-channel leakage. Our proposed Timing Side Channel Sensitive Program Obfuscation Optimization Framework (TSC-SPOOF) determines the combination of obfuscation transformation functions that produce optimized obfuscated programs with preferred optimization parameters. In particular, TSC-SPOOF employs normalized compression distance (NCD) and channel capacity to measure obscurity and timing side-channel leakage, respectively. We also use RISC-V rocket core running on a Xilinx Zynq FPGA device as part of our framework to obtain realistic results. The experimental results clearly show that our proposed solution leads to cryptographic programs with lower execution time, higher obscurity, and lower timing side-channel leakage than unguided obfuscation.

CCS Concepts: • Security and privacy → Software reverse engineering; • Software and its engineering → Software reverse engineering; • Computer systems organization → Embedded software;

Additional Key Words and Phrases: Obfuscation, optimization, NCD, LLVM, timing side-channel, channel capacity, FPGA

ACM Reference format:
Arnab Kumar Biswas. 2021. Cryptographic Software IP Protection without Compromising Performance or Timing Side-channel Leakage. ACM Trans. Archit. Code Optim. 18, 2, Article 20 (February 2021), 20 pages. https://doi.org/10.1145/3443707

1 INTRODUCTION

Different types of embedded systems need to use various types of cryptographic software intellectual property (IP) from different vendors. These vendors need to consider software IP theft seriously while selling the IP. Program obfuscation is an effective technique to hide internal implementation details of the code from an adversary [37] who can perform a reverse engineering attack. However, program obfuscation often involves combining different transformation
functions in an arbitrary fashion, which could still result in large code size, performance degradation, and even timing side-channel leakage. Timing side-channel leakage indicates information leakage in timing domain that can be used by the attacker to gain sensitive information (like secret key) from the running program [7]. An attacker can also use this attack to obtain certain information to reduce the effort required in reverse engineering attack. For example, an attacker can obtain information such as instruction count, raised interrupt count, accessed instruction addresses, and accessed memory addresses from timing side-channel leakage. Kocher et al. first reported reverse engineering attack using side-channel leakage in Reference [23]. Later, a side-channel-based disassembler is also reported in Reference [16].

### 1.1 Motivational Example

As a motivational example, Figure 1 shows the impact of randomly selected obfuscation transformation sequences (seq_1, ..., seq_5) on the timing information leakage of the ModExp application compared to the original code (baseline). The information leakage is measured using channel capacity metric using LeakiEst tool [9] (further details are described later). It can be observed that the choice of transformation sequences has a notable impact on the information leakage. In addition, certain transformation sequences (e.g., seq_3 and seq_4) can result in higher information leakage than the original program. Also note that though seq_5 is providing a promising solution for this program, this particular transformation sequence will not provide similar promising behavior for other programs (this will be shown later) and finding this kind of sequence for a program is not a trivial task.

So it is clear from the above example that there is no certainty that every instance of same obfuscated program will have same information leakage property. Sometimes, the information leakage is higher and other times it is lower and we do not have any control over it. Same observation can be made for the code size and performance of the program also. To address this problem, in this article, we present a genetic algorithm (GA)-based Timing Side Channel Sensitive Program Obfuscation Optimization Framework (TSC-SPOOF) to find the permutation and combination of obfuscation transformation functions that produce codes that are optimal with respect to performance, reverse engineering protection, and timing side-channel leakage. As we are mainly targeting software security in embedded systems, we use RISC-V based Rocket core execution environment running on...
FPGA board as part of our proposed TSC-SPOOF framework. We use the execution times obtained from this environment to estimate the timing side-channel leakage in terms of channel capacity metric. To the best of our knowledge, this is the first work that proposes a multi-objective optimization approach for program obfuscation that simultaneously optimizes for performance, obscurity, and timing side-channel leakage. Note that we use the term obscurity to denote the effectiveness of program obfuscation (i.e., the difficulty in reverse engineering), which is the only objective of common obfuscation methods and we use the Normalized Compression Distance (NCD) metric [13] to measure obscurity.

The main contributions of this work are as follows:

1. We show that normal obfuscation methods do not produce obfuscated code with desired properties, and there is a need to address this problem.
2. We propose an optimization framework called TSC-SPOOF that produces code with optimized performance, obscurity, and timing side-channel leakage.
3. Our framework uses an embedded system execution environment (RISC-V based processor on FPGA) as part of the framework to get realistic results.
4. Experimental results clearly show the effectiveness of the framework and also show that TSC-SPOOF is configurable if we choose to optimize only two parameters.

The remainder of the article is organized as follows: Section 2 presents a brief overview of related work and their limitations. Section 3 describes the threat model considered in this work and also provides the necessary background for our proposed TSC-SPOOF framework. Section 4 describes the proposed TSC-SPOOF framework with a detailed description of the genetic algorithm incorporated for solving our optimization problem. Experimental setup and results are given in Section 5, while Section 6 concludes the article.

2 RELATED WORK

There exists a lot of work in both industry and academia to increase the software IP security through obfuscation [15]. However, most of the existing works in software obfuscation do not address the tight interplay between code obfuscation, performance, and information leakage.

The authors in Reference [25] have presented a method to determine a sequence of obfuscation transformations that leads to optimal obscurity. They have introduced an obscurity language model to assess the obfuscation effectiveness. The authors have also proposed a guided stochastic algorithm based on Markov chain Monte Carlo method to search for the optimal sequence. Their work focuses on source-to-source transformation for JavaScript, and the program execution time is not considered. Unlike our work, the authors in Reference [25] have not examined the effect of obfuscation on timing side-channel leakage. The work in Reference [42] has addressed the software obfuscation problem on the Android-based mobile computing systems. They have proposed methods to optimize the obfuscation and software execution time, and software complexity metrics are used to evaluate the obfuscation level. Also, they have not considered side-channel security in their work.

Another related work albeit in a different field is Reference [40], where the authors have considered the security and real-time performance of a networked control system. They have developed a co-evolutionary genetic algorithm to optimize for performance and security on a networked DC motor system. A security metric that considers the vulnerability of encryption algorithms to brute force attacks and also the system oscillation level is utilized. They have also targeted their framework to a distributed networked control system in Reference [41].

Works exist in literature that do not target obfuscated code but have proposed optimization solution to optimize performance and security parameters. Authors in Reference [31] have proposed
a framework in infrastructure as a service (IaaS) cloud to optimize the cloud resources and security provided to users. They have mainly targeted the dynamically instantiated virtual machines (VMs) to optimize the security and resources allocated to the VMs. They have used the percentage of CPU utilization, percentage of memory utilization, execution time of programs, security measure, and the hourly cost of VMs. They have implemented a randomized algorithm to perform the optimization.

The authors in Reference [34] have introduced a compiler that transforms (obfuscates) the program to protect against side-channel attacks. Their solution is able to execute extraneous program paths called decoy paths and uses control flow obfuscation and software ORAM to hide memory access traces. Their solution is only targeted to SGX-enabled Intel processors. In their solution, both real and decoy paths execute actual program instructions and both paths are allowed to update memory, which helps to keep the decoy path hidden. Also, the user needs to annotate the hidden variables for taint analysis. The work in Reference [19] proposed control flow obfuscation where the control flow of a program is dynamically disrupted by converting its conditional branches randomly. This conversion causes changes in execution times and makes the code unintelligible for reverse engineering and increases the difficulty of side-channel attacks. The proposed work in Reference [19] will be more effective if it uses our proposed optimization technique in addition to their control flow obfuscation method. Their obfuscation method may not always provide intended reduced timing side-channel leakage as shown in our motivational example. In addition to low timing side-channel leakage, our technique can provide low program execution time and high level of obfuscation irrespective of program.

Authors in Reference [39] have proposed a security system called BUNSHIN that allows different security methods to be combined to protect a program without reducing its performance. In their solution, multiple existing runtime security checks are done without any conflict in different program variants. They generate program variants that are vulnerable to various attacks and also are secure to different attacks. That means each variant functionality is similar in typical situations and they behave differently under attack. Then these variants are run in parallel in a synchronized manner. This solution is not suitable for systems with limited hardware resources, because the BUNSHIN solution simply executes program variants on multiple processor cores in parallel without sacrificing security and performance.

It is reported in Reference [6] that software only virtual black box (VBB) obfuscation of general programs is impossible. But authors in Reference [32] have proposed a system called HOP to implement VBB obfuscation of RAM programs using secure hardware to circumvent previous impossibility results. They use oblivious hardware RAM, scratchpad memories, instruction scheduling techniques, and context switching. HOP processor is implemented with a secret key. This processor is with a receiver who is also the attacker and runs the obfuscated programs. The key is with the sender, who uses it to obscure the program to prevent the attacker from learning anything from program’s timing or memory access pattern. The security of the whole solution depends on the secret key in the HOP processor.

None of the existing works in literature have investigated the impact of obfuscation on performance and timing side-channel leakage at the same time. In our work, we present a GA-based framework to obtain a combination of obfuscation transformation functions that simultaneously optimize code performance, obscurity, and timing side-channel security.

3 PRELIMINARIES

In this section, we describe the threat model considered in this work and also provide the necessary background for our proposed TSC-SPOOF framework. More specifically, we describe about program obfuscation, software complexity metric, and timing side-channel capacity.
Table 1. Transformation Functions of LLVM-obfuscator

| Instruction substitution | Bogus control flow | Control flow flattening |
|--------------------------|--------------------|-------------------------|
| -sub                     | -bcf               | - Ilia                  |
| -sub_loop = 2 to 10      | -bcf_loop = 2 to 10| -split                  |
|                          | -bcf_prob = 1 to 99| -split_num = 2 to 10    |

3.1 Threat Model

In this article, we mainly consider reverse engineering attacks on software IP to extract information about the software itself like the implementation details. The main solution available in literature is program obfuscation, as discussed later. We also consider this solution against reverse engineering attack but a serious shortcoming of existing solutions is that they do not consider another serious vulnerability called timing side-channel leakage. In this work, we consider this vulnerability of software IP that is supposed to be secure, i.e., obfuscated software IP. Utilizing timing side-channel attack, an attacker observes timing behavior of the software IP to infer secret information like secret key used in the program. We show that our proposed TSC-SPOOF framework can produce obfuscated software IP that has better performance and timing side-channel leakage behavior compared to a random obfuscated code. Further discussion on timing side-channel is given later.

3.2 Program Obfuscation

A formal definition of source code obfuscation can be given as follows: An obfuscator $O$ is defined as a compiler that converts a source code $S$ into its obfuscated version $O(S)$, which is functionally equivalent to $S$ yet is unintelligible to an attacker to retrieve $S$ from $O(S)$. There are mainly two types:

1. **Source-to-source obfuscation**: This type of obfuscation tool converts the source code into a complicated form to prevent an attacker to perform reverse engineering attack. The obfuscator uses various techniques such as mangling of the integers, striping spaces, striping newlines, bogus control flow insertion, and control flow flattening to obfuscate the source code. A few examples of source-to-source obfuscators are Tigress [12], Sandmark [11], and Stunnix [1].

2. **Source-to-binary obfuscation**: This type of obfuscation tool is a compilation suite integrated with obfuscation transformation to generate obfuscated binary code. It is assumed that the attacker does not have the access to the source code. Obfuscation transformations such as instruction substitution, bogus control flow insertion, control flow flattening, and basic block splitting are commonly used. Though the functionality is not affected by the alterations, the transformation functions and their order impacts the performance (execution time) and the obscurity level of code [14]. In this article, we mainly target this type of obfuscator and use Obfuscator-LLVM (O-LLVM) [22] for our experiments. O-LLVM [2] is mainly built as different passes where these passes work on LLVM intermediate representation (IR) code that is language-agnostic and independent of the target architecture. LLVM is a collection of modular and reusable compiler and toolchain technologies. It is the full name of the project (not an acronym), which contains subprojects like Clang.

O-LLVM mainly supports three transformation functions called instruction substitution, bogus control flow, and control flow flattening, as shown in Table 1. Each of the transformation functions can be fine-tuned through additional parameters: The loop value (-sub_loop, -bcf_loop, and -split_num) determines how many times the respective function will be applied to the input code. We consider loop values of 2 to 10, because this provides a wide range of loops. A probability...
can also be supplied to the obfuscator to control the density of the alterations. We consider the probability of 1 to 99, because 0 or 100 are certain values. It is highly unlikely that a randomly chosen combination will provide optimized results. As an example, Figure 2 shows the plot of the obfuscation levels in terms of NCD (described later) values and the normalized execution time for which the obfuscation transformation functions are chosen randomly. Normalized execution time is defined as the relative increase of execution time with respect to un-obfuscated/original code’s execution time. If the obfuscation level (NCD value) is closer to 1, the obfuscated code is more dissimilar to the original source code, making it more difficult for an attacker to relate portions of the code to its origin. As it can be seen in the figure, the execution time and the obfuscation level vary for different benchmarks chosen. For example, although MD5 has a higher obfuscation level, the specified transformation sequence is not advisable to obfuscate the program due to a significant increase in the execution time.

So, it is clear that the user does not know which permutation and combination of the transformation functions will give the desired result in terms of execution time and obscurity. Considering that a transformation can be applied multiple times (e.g., the instruction substitution is followed by the insertion of dead code and bogus flow control, which is again succeeded by an additional round of substitutions), the user has to select, explore, and evaluate myriad permutations for every single source code to find the best combination. In addition, taking into account the prevention of timing side-channel attack vulnerability, this task might take a very long time even for a simple application.
3.3 Software Complexity Metric or Obscurity of Code

Software complexity metric or obscurity of code denotes the effectiveness of obfuscation by quantifying the amount that the program has changed. Normalized Information Distance (NID) [38] is known to measure code obfuscation resilience and effectiveness of obfuscation methods by measuring relative similarity between strings $x$ and $y$. It is defined as,

$$NID(x, y) = \frac{\max(K(x|y), K(y|x))}{\max(K(x), K(y))},$$

where $K(x|y)$ is algorithmic information or Kolmogorov complexity of $x$ given $y$ as input. But the problem with this metric is that it is not computable and hence it is approximated by NCD metric using normal compressor [13]. Works presented in References [27–30] have performed extensive analysis of NID and its approximate value using NCD to measure obscurity of code after obfuscation. NCD takes as arguments two objects (literal files) and evaluates a formula expressed in terms of compressed version of the objects separately and both combined. Let $x$, $y$, and $xy$ denote the binary files generated from RISC-V LLVM compilation for unobfuscated code, obfuscated code, and concatenated or combined code $(x||y)$, respectively. If $Z(x)$, $Z(y)$, and $Z(xy)$ denote the binary length of the files $x$, $y$, and $xy$, respectively, after compressed by a compressor $Z$, then according to Reference [13],

$$NCD_z(x, y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}}.$$  

We use bzip2 compressor to obtain the NCD values and the NCD value ranges from 0.0 (maximally similar) to 1.0 (maximally dissimilar).

3.4 Timing Side-channel Capacity

A timing side channel is a communication channel created by unintentional information leakage by victim program. An attacker exploits this leakage to extract the secret key by observing execution time of the cryptographic program. This means that an attacker is assumed to be able to observe the execution time of the victim program but does not have access to the actual program code or secret key. Our aim is to obtain the upper bound (channel capacity) of information transmission rate through the timing side channel to identify the side-channel attack’s effectiveness in numerical terms. Timing side-channel leakage is sometimes unintentionally introduced during the compiler optimization stage. Apart from thwarting the reverse engineering attack, our work also studies the effect of obfuscation on timing side-channel leakage. In fact, by efficiently choosing the obfuscation transformation functions through our TSC-SPOOF framework, timing side-channel leakage can be reduced without sacrificing other important properties of the code.

We use channel capacity metric to measure the timing side-channel leakage, and we use a tool called LeakiEst [9] for this purpose. Channel capacity is the rate at which information can be reliably transmitted over a communication channel, which is the timing side channel in our case. The tool uses iterative Blahut-Arimoto algorithm [4, 8] to calculate the channel capacity. Note that our aim is not to restrict the output code to have only absolute value of 0 channel capacity or no leakage but to reduce the leakage as close to 0 as possible.

4 PROPOSED TSC-SPOOF FRAMEWORK

This section provides a detailed description of the proposed TSC-SPOOF framework using genetic algorithm to address the obfuscation optimization problem.
4.1 Problem Formulation

Before describing the TSC-SPOOF framework, let us first define the optimization problem and the cost function. The optimization problem is to find a set of obfuscation transformation and their order from a pool of different transformation functions. We assume a given number of obfuscation transformation functions \( \{F_1, F_2, \ldots, F_N\} \), with each function consisting of multiple sub-functions:

\[
F_1 = \{f_{11}, f_{12}, \ldots, f_{1m}\}, \\
F_2 = \{f_{21}, f_{22}, \ldots, f_{2n}\}, \\
\vdots \\
F_N = \{f_{N1}, f_{N2}, \ldots, f_{Nm}\}
\]

and each sub-function’s cardinality varies from 1 to \( M \), where \( M \) can be same or different for each of the sub-functions. For a particular application, we want to find a permutation and combination of transformation functions that results in a binary code with (1) minimum execution time, (2) maximum obscurity value, and (3) minimum leakage channel capacity. So, given the execution time of the application \( E(P) \), complexity metric \( NCD(R) \), and the leakage channel capacity \( C(S) \), the cost of the overall optimization problem \( C_{PRS} \) can be formulated as:

\[
C_{PRS} = \alpha * E(P) - \beta * NCD(R) + \gamma * C(S),
\]

where \( \alpha \), \( \beta \), and \( \gamma \) are the weights of the respective parameters each ranging from 0 to 1. The subscripts P, R, and S refer to performance, reverse engineering, and security, respectively.

4.2 Main Components of the TSC-SPOOF Framework

Figure 3 shows our genetic algorithm based TSC-SPOOF framework. The framework mainly consists of LLVM obfuscator that generates obfuscated binary code and also can generate non-obfuscated/original binary code. As we are mainly targeting embedded systems, we decide to use RISC-V based Rocket core execution environment running on FPGA as part of the framework. The gray boxes indicate the core stages involved in the genetic algorithm. In the first step of population generation, obfuscation transformation functions are chosen randomly with different permutation and combination. Figure 4 shows the chromosome model of the optimization problem. Here, each obfuscation transformation is considered as a gene and genes are combined to form a chromosome. If we consider all the permutations given in Table 2, theoretically, myriad chromosomes can be constructed. Each chromosome represents a compilation flag. The length of the chromosomes varies from 1 to 6. The generated population (pool of compilation flags) is used in RISC-V LLVM environment to compile a source code. Note that we generate both obfuscated and non-obfuscated binary files where the non-obfuscated binary file generation does not require any obfuscation compilation flag and is only used to calculate the NCD value. The obfuscated binary file is executed on top of a Linux OS running on a RISC-V-based Rocket core microprocessor platform that is running on a Zedboard FPGA.

Algorithm 1 summarizes the entire procedure of our genetic algorithm. The inputs to the algorithm are the size of the population \( N \) and the obfuscation transformation functions. Currently, we use a population size of 400 in our implementation. The output is a pool of best combination of obfuscation transformations. First, a population is initialized by generating chromosomes randomly. Next, using Equation (3), the algorithm calculates the fitness value of each genotype (line 2). Then the search process is carried out until the termination criterion is met. The termination criteria used in this work is based on the number of iterations for which a better solution is still unreachable. Genetic operations of crossover and mutation are carried out in each iteration to explore and exploit the search space (lines 6 and 7). Again the fitness value of each chromosome/genotype of a newly generated population are calculated and using the selection operation (line 9) new chromosomes are selected for next generation. Further details of the genetic operations are given next.
4.3 Genetic Operations

The three main steps involved in genetic operations are selection, crossover, and mutation, which are described here in more detail.

4.3.1 Selection. This is the process to form a new population by choosing chromosomes that have better fitness value in the old population. Through the selection procedure, the overall quality of the population can be enhanced. This can be achieved by incorporating either Roulette wheel selection or tournament selection approach. In the roulette wheel approach, an imaginary 360-degree wheel is spun and the chromosome to which it points to is selected. This process is repeated until the population is filled. Typically, roulette wheel selection technique is used in the problems where the objective function needs to be maximized [5]. This is because, for a minimization problem, the genetic algorithm converges prematurely in roulette wheel selection approach. On the contrary, in
Algorithm 1: Genetic Algorithm for Obfuscation Transformation Exploration

**Input:** Obfuscation transformation functions, also called compilation flags

**Output:** Pool of best combination and permutation of obfuscation transformation functions

1. Generate random population
2. Fitness evaluation of each genotype
3. while Termination criterion is not met do
   4. for $j = 0 \rightarrow N$ do
      5. $\{P_1, P_2\} =$ random_selection();
      6. Crossover($P_1$, $P_2$);
      7. Mutation($Chromosome$, $PM$); ▷ Algorithm 3
     end for
   8. Genotype evaluation and selection for next round; ▷ Algorithm 2
   9. end while
10. return Pool of best combination of obfuscation transformation functions obtained;

Tournament selection approach, a number of individuals are randomly picked (in our case, 2) from the population and the individual with the best fitness value compared to the counterpart is chosen as one of the parents for the next round. Since our objective function needs to be minimized, we have adopted tournament selection approach for building a new population.

**Algorithm 2:** Genetic Selection through Tournament Selection Approach

**Input:** Population $P_{pop} = \{C_1, C_2, \ldots, C_N\}$ generated in Random Population Generation Stage and their fitness values ($fit(C_1), fit(C_2), \ldots, fit(C_N)$)

**Output:** Selected Chromosomes (CH)

1. CH = $\emptyset$
2. for $i \leftarrow 1$ to $N$ do
3.   Pick a random chromosome $C_{rand}$ from $P_{pop}$ in which $C_i \neq C_{rand}$;
4.   if $fit(C_i) < fit(C_{rand})$ then
5.     Champion = $C_i$;
6.   else
7.     Champion = $C_{rand}$;
8.   end if
9.   CH = CH $\cup$ Champion;
10. end for
11. return Pool of Champion chromosomes (CH);

Algorithm 2 demonstrates our selection procedure using tournament selection process. The algorithm begins with the first individual of the population and randomly selects another individual from the population (line 3). Fitness values of the individuals are compared to make the selection (lines 4–7). The individual with the lowest fitness value among the two will be selected as a champion. This procedure is iterated for all the chromosomes in the population, and winners are
Table 2. An Example of Our Tournament Selection Approach

| Chromosome | Competitor | Fitness Competition | Champion |
|------------|------------|---------------------|----------|
| C₁         | C₄         | 1,303.6 < 1,155.2 (No) | C₄       |
| C₂         | C₃         | 1,157.6 < 1,164.3 (Yes) | C₃       |
| C₃         | C₂         | 1,164.3 < 1,157.6 (No) | C₂       |
| C₄         | C₁         | 1,155.2 < 1,303.6 (Yes) | C₁       |

scheduled for further steps, i.e., crossover and mutation. An example of our tournament selection approach is shown in Table 2. To ensure diversification and avoid redundancy in the generations, we use twin removal as proposed in Reference [21].

4.3.2 Crossover. Crossover mainly aims to obtain the individual with higher fitness value. In the literature, it can be found that there is a tendency to use chromosomes of predetermined fixed length. This makes it easy for crossover to choose identical crossover points in each parent chromosome and to form offspring with homologous features. The challenge arises when two parent genotypes of variable lengths have to be crossed, so resulting offspring genotypes are logically interpretable and have meaningful constituents from both parent chromosomes. Adopting the same crossover technique for variable length genotypes might result in deformed or absurd offspring. The problem of crossover for variable length genotypes is stated as follows:

Given a randomly selected crossover locus in one parent chromosome, how to find an appropriate locus to break the other parent chromosome of variable length to exchange homologous sections as far as possible?

We propose modified SAGA algorithm (Algorithm 3) to solve this problem, which is an extension of SAGA [17, 18] framework. Our proposed Algorithm 3 accepts parents P₁ and P₂ of lengths m and n, respectively, and outputs X, a best candidate for crossover point in P₂. Lines 3–7 make sure that there is no repetition of transformation function in both P₁ and P₂. In the original SAGA framework, the crossover point is chosen randomly. In our case, the midpoint of P₁ is chosen as the crossover point (line 19). Two vectors L and R are used to store the outputs from two separate calls to Hirschberg’s algorithm [20]. In line 20, Hirschberg’s algorithm is applied to the left-hand sequence of parent P₁ (i.e., from 1 to c) and to the whole sequence of P₂ (i.e., from 1 to n). The Hirschberg’s algorithm returns the best matching scores with the left-hand sequence of P₁ until c. Similarly, the algorithm is applied to the right-hand sequence of P₁ and the whole of P₂ in reverse order (i.e., starting from right end) in line 21. The for loop (lines 23–29) evaluates each possible crossover point j in the parent chromosome P₂. The sum of LCSS for both the segments of P₁ is temporarily stored in variable t. The value of t will either increase or remain constant until it reaches a global maximum, and after that the value of t will decrease with occasional outstretches. The algorithm returns the best crossover point X in the parent chromosome P₂, i.e., the best left and right LCSS of P₂.

4.3.3 Mutation. In genetic algorithm, mutation is used to provide exploration, and crossover is used to enable the population to converge at one of the better solutions found so far (i.e., exploitation). Choice of the mutation rate is very important, because choosing very large or very small mutation rate will either prevent the algorithm to converge quickly or cause premature convergence. Hence, we decide to use a mutation probability PM that is problem specific and dynamic (changes in every iteration) [33] as given by

\[
PM = \frac{\delta m(FT_{max} - FT_{bestM})}{FT_{max} - FT_{avg}}
\]
ALGORITHM 3: Modified SAGA Crossover Algorithm for Variable Length Chromosomes

Input: Parents $P_1$ and $P_2$
Output: Crossover point $X$, best candidate for a crossover point in $P_2$

1: $m = |P_1|$
2: $n = |P_2|$
3: if $n=0$ or $m=0$ then
4: return 0
5: else if $n=1$ and $m=1$ then
6: if $P_1(0) = P_2(0)$ then
7: return 1
8: else
9: return 0
10: end if
11: else if $m=1$ then
12: for $i ← 0$ to $n$ do
13: if $P_1(0) = P_2(i)$ then
14: return 1
15: end if
16: end for
17: return 0
18: else
19: $c ← \lceil \frac{m}{2} \rceil$; \Comment{Crossover point in $P_1$}
20: $\text{Hirschberg}_{\text{LCSS}}(1, c, 1, n, P_1, P_2, L)$;
21: $\text{Hirschberg}_{\text{LCSS}}(m, c-1, n, 1, P_1, P_2, R)$;
22: $\text{max} ← 0$
23: for $j ← 1$ to $n$ do
24: $t ← L(j) + R(n-j)$;
25: if $t > \text{max}$ then
26: $\text{max} ← t$;
27: $X ← j$;
28: end if
29: end for
30: return $X$;
31: end if

where $FT_{\text{max}}$ is the maximum fitness value in the population, $FT_{\text{bestM}}$ is the fitness value of the chromosome to be mutated, $FT_{\text{avg}}$ is the average fitness value of the current population, and $\delta_m$ is a positive constant less than 1. Currently, we use $\delta_m = 0.02$ in our implementation.

For every offspring generated from the crossover operation, the genetic mutation is performed with probability $PM$, as depicted in Algorithm 4. A gene (transformation function) of the chromosome selected randomly is flipped (line 5) where flipping denotes replacing a transformation function (gene) of the chromosome with another unique transformation function. Figure 5 summarizes the mutation operation.

5 EXPERIMENTAL RESULTS

In this section, we describe the experimental setup to test our TSC-SPOOF framework and, later, we discuss the obtained experimental results.

5.1 Experimental Setup

On the hardware side, we use a RISC-V-based Rocket core running on Zedboard that contains the Zynq-7000 XC7Z020-CLG484-1 FPGA device. On the software side, the binary file generated
**Algorithm 4:** Genetic Mutation

**Input:** Chromosome $C$, Mutation rate $PM$

**Output:** New mutated chromosome

1. $m = |C|$  \[\text{Length of the chromosome}\]
2. $i = \text{int}(\text{rand}(m))$;  \[\text{Random number between 1 to m}\]
3. $RN = \text{rand}();$  \[\text{Random number between 0 to 1}\]
4. if $RN \leq PM$ then
5. $C(i) = \text{Replace the transformation function with another transformation function chosen randomly};$
6. end if
7. return the new mutated chromosome $C_{\text{new}}$;

Using RISC-V O-LLVM+gcc is executed on Linux on the Zedboard to measure the execution times using front-end server (fesvr) proxy kernel (pk), a lightweight application execution environment. Execution time of a program is measured by monitoring the control and status register of the rocket chip called CYCLE using RDCYCLE instruction in terms of the number of clock cycles executed by the rocket chip. For estimating timing side-channel leakage using LeakiEst, we use two distinct secret keys to obtain the execution times. Each sample of execution time is annotated with a Boolean variable to indicate if the sample is resulted from the first or the second secret key. We collect 1,000 samples for each of the two secret inputs for each evaluated program. The channel capacity value varies from 0 (no leakage) to 1 (complete leakage). Please note that this 1-bit information reveals which secret input is used among the two. Authors in [10] have also shown that LeakiEst can successfully obtain the channel capacity of the timing side-channel leakage using this method.

We use two sets of benchmarks (in C) to test our TSC-SPOOF solution. The first set of benchmarks are taken from Reference [3] and consists of AES, MD5, SHA-1, and blowfish and is used for optimizing two parameters of the objective function: performance and obscurity (by setting $\gamma = 0$ and varying $\alpha$ and $\beta$ in Equation (3)). The main reason of doing this experiment is to show that the TSC-SPOOF is configurable and we can select which parameters (only two instead of all three) we want to optimize for any program. The second set of benchmarks is taken from Reference [26]. This set consists of (i) square-and-multiply modular exponentiation (ModExp) from RSA to encrypt and decrypt a message [36] and (ii) modular multiplication (MulMod16) from the IDEA cipher [24]. We use these programs to test our TSC-SPOOF framework by optimizing all three
parameters of the objective function, i.e., performance, obscurity, and timing side-channel leakage capacity. We run the genetic algorithm to get the results on a PC with Intel core i5 processor with 2 GB RAM and execution time varies from 13 minutes to 27 minutes for different benchmarks.

5.2 Results Discussion

Figure 6 shows the scatter plot obtained after the termination criteria for the GA is met, using our TSC-SPOOF framework for AES, SHA-1, blowfish, and MD5, respectively. In each figure, the black triangles indicate the genotypes (transformation sequence/compilation flags) that are finally selected for obfuscating the program. We can clearly observe that our TSC-SPOOF framework generates points that are optimized for both performance (normalized execution time with respect to the non-obfuscated baseline program) and obscurity (NCD value).

We also perform an experiment to see the effects of $\alpha$ and $\beta$ in Equation (3) on the cost function values. Figure 7 shows that average normalized cost function values of AES optimized points do not change much for different $\alpha$ and $\beta$ values. Each point in this plot is obtained after keeping the weights fixed to a value (keeping $\alpha = \beta$) to obtain the set of optimized points (similar to black triangle points in Figure 6(a)). After obtaining the optimized points, we obtain the average of their cost function values and divide this average value by the maximum cost function value in the initial population to get the average normalized cost function value. This is repeated for all weight values from 0.1 to 1. We also observe similar behavior for other benchmarks.

ACM Transactions on Architecture and Code Optimization, Vol. 18, No. 2, Article 20. Publication date: February 2021.
Figures 8 and 9 show the experimental results of optimization considering all combinations of two parameters and also considering all three parameters for ModExp and MulMod16 programs to study the impact of obfuscation on timing side-channel leakage. Figures 8(a) and 9(a) are similar to Figure 6. The rest of the sub-figures are the plots of different combination of the parameters of the objective function (Execution time, NCD, and channel capacity). We use the term channel capacity to denote leakage channel capacity only. Our proposed TSC-SPOOF framework provides the degree of freedom to optimize any of the two parameters of the objective function or all three. Table 3 shows the values of objective function parameters of some selected optimized points mainly considering two parameters at a time. Optimized points of all three parameters are only given for ModExp and MulMod16 programs (i.e., second set of benchmarks). The main aim of optimization is to reduce execution time, maximize NCD, and reduce channel capacity. It can be clearly seen that maximum performance overhead (normalized execution time) is at an acceptable level of 0.385. In most of the cases, NCD value is more than 0.7, which is a good indication of obscurity, i.e., difficulty in reverse engineering. The channel capacity varies from 0.0104 to 0.1705, which is a good indication of minimal timing side-channel leakage. There is no threshold (upper bound for NCD and lower bounds for normalized execution time and channel capacity) to choose the optimized points. But we have to make sure that NCD value should be as high as possible, and normalized execution time and channel capacity must be as low as possible. We observe that the points that have NCD value, normalized execution time, and channel capacity greater than 0.7, less than 0.3, and less than 0.2, respectively, are better-optimized points. We also observe that the obfuscation transformation combinations to optimize a program are different for different programs.

We also use dudect tool [35] to verify that our proposed TSC-SPOOF framework gives optimized points with reduced timing side-channel leakage for ModExp and MulMod16 programs. We
observe that for optimized ModExp and MulMod16 programs, around 1M and 2M measurements are required, respectively, to cross the t statistic value of 10. But it requires only 20K measurements for both programs in the initial population to cross the same t statistic value. So, we can verify that our TSC-SPOOF framework indeed provides optimized points with reduced timing side-channel leakage.

The proposed framework is compared with previous works in Table 4. It can be clearly observed that only our proposed work optimizes all the three properties of a program, i.e., performance, obscurity, and side-channel security.

6 CONCLUSION

Obfuscation plays a significant role in defending a cryptographic software IP against reverse engineering attacks by adopting various code transformation techniques. In this work, we have proposed a genetic algorithm–based framework called TSC-SPOOF to optimize the cryptographic program performance, obscurity, and timing side-channel leakage. Experimental results show that TSC-SPOOF can find transformation sequences for each of the benchmark programs. Although
Fig. 9. Scatter plot of MulMod16 benchmark showing results of optimization considering all two parameter combinations and also considering all three parameters. Note that we use the term channel capacity to denote the leakage channel capacity only.

Table 3. Values of Objective Function Parameters of Some Selected Optimized Points Mainly Considering Two Parameters at a Time

| Benchmark Program | Execution Time (Normalized) | NCD  | Channel Capacity |
|-------------------|-----------------------------|------|------------------|
| AES               | 0.2264                      | 0.79 | -                |
|                   | 0.2368                      | 0.782| -                |
|                   | 0.245                       | 0.8  | -                |
| SHA-1             | 0.2598                      | 0.82 | -                |
|                   | 0.205                       | 0.79 | -                |
|                   | 0.0779                      | 0.619| -                |
|                   | 0.3256                      | 0.794| -                |

(Continued)
Table 3. Continued

| Benchmark Program | Execution Time (Normalized) | NCD | Channel Capacity |
|-------------------|-----------------------------|-----|-----------------|
| blowfish          |                             |     |                 |
|                   | 0.271                       | 0.7 | -               |
|                   | 0.2954                      | 0.747 | - |
| MD5               |                             |     |                 |
|                   | 0.163                       | 0.783 | - |
|                   | 0.1106                      | 0.684 | - |
|                   | 0.112                       | 0.6814 | - |
|                   | 0.2567                      | 0.8031 | - |
|                   | 0.2512                      | 0.7685 | - |
|                   | 0.198                       | 0.7647 | - |
|                   | 0.104                       | -    | 0.1206          |
|                   | 0.1684                      | -    | 0.1199          |
|                   | 0.1825                      | -    | 0.0633          |
| ModExp            |                             |     |                 |
|                   | -                           | 0.767 | 0.1752         |
|                   | -                           | 0.789 | 0.1619         |
|                   | -                           | 0.7412 | 0.0911 |
|                   | 0.1368                      | 0.708 | 0.059           |
|                   | 0.2914                      | 0.741 | 0.0911          |
|                   | 0.2578                      | 0.715 | 0.1503          |
| MulMod16          |                             |     |                 |
|                   | 0.28                        | 0.7686 | - |
|                   | 0.156                       | 0.729 | -               |
|                   | 0.134                       | 0.703 | -               |
|                   | 0.255                       | -    | 0.0235          |
|                   | 0.205                       | -    | 0.0993          |
|                   | 0.135                       | -    | 0.1705          |
|                   | -                           | 0.817 | 0.0144         |
|                   | -                           | 0.793 | 0.0569         |
|                   | -                           | 0.815 | 0.197          |
|                   | 0.25                        | 0.7012 | 0.0104       |
|                   | 0.385                       | 0.7829 | 0.024     |
|                   | 0.278                       | 0.727 | 0.1265          |

Optimized points of all three parameters are only given for ModExp and MulMod16 programs.

Table 4. Comparison with Related Works

|                | Performance optimization | Obscurity optimization | Security optimization (Side-channel security) |
|----------------|--------------------------|------------------------|----------------------------------------------|
| [25]           | No                       | Yes                    | No                                           |
| [42]           | Yes                      | Yes                    | No                                           |
| [31, 39–41]    | Yes                      | No                     | Yes (No)                                     |
| [32, 34]       | No                       | No                     | Yes (Yes)                                    |
| Proposed work  | Yes                      | Yes                    | Yes (Yes)                                    |

this work has targeted the optimization of source-to-binary obfuscation process, it can be easily ported to source-to-source obfuscation transformation process, as the transformation functions are similar.
REFERENCES

[1] Stunix C/C++ Obfuscator. 2020. Retrieved from http://stunix.com/support/doc/cxxo/cxx-obfus.shtml.
[2] Obfuscator-LLVM. 2020. Retrieved from https://github.com/obfuscator-llvm/obfuscator/.
[3] Crypto-algorithms benchmark. 2020. Retrieved from https://github.com/B-Con/crypto-algorithms.
[4] S. Arimoto. 1972. An algorithm for computing the capacity of arbitrary discrete memoryless channels. IEEE Trans. Inf. Theor. 18, 1 (Jan. 1972), 14–20.
[5] Rajesh Kumar Arora. 2009. Optimization (1st ed.). Chapman and Hall/CRC.
[6] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2012. On the (im)possibility of obfuscating programs. J. ACM 59, 2 (2012).
[7] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. 2017. A survey of timing channels and countermeasures. ACM Comput. Surv. 50, 1 (Mar. 2017). DOI: https://doi.org/10.1145/3023872.
[8] R. Blahut. 1972. Computation of channel capacity and rate-distortion functions. IEEE Trans. Inf. Theor. 18, 4 (July 1972), 460–473.
[9] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. Springer, 690–695.
[10] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A tool for estimating information leakage. In Computer Aided Verification, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin, 690–695.
[11] Christian Collberg, Ginger Myles, and Andrew Huntwork. 2003. Sandmark–A tool for software protection research. IEEE Security and Privacy 1, 4 (July 2003), 40–49. DOI: https://doi.org/10.1109/MSEP.2003.1219058
[12] The TIGRESS Obfuscator. 2020. Retrieved from https://tigress.wtf/download.html.
[13] R. Cilibrasi and P. M. B. Vitanyi. 2005. Clustering by compression. IEEE Trans. Inf. Theor. 51, 4 (2005), 1523–1545.
[14] Christian Collberg and Jasvir Nagra. 2009. Surreptitious Software Obfuscation, Watermarking, and Tamperproofing for Software Protection (1st ed.). Addison-Wesley Professional.
[15] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations.
[16] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. 2010. Building a Side Channel Based Disassembler. Springer Berlin, 78–99.
[17] I. Harvey. 1992. SAGA cross: The mechanics of recombination for species with variable-length genotype. In Parallel Problem Solving from Nature 2. North-Holland, 269–278.
[18] I. Harvey. 1992. Species adaptation genetic algorithm: The basis for a continuing SAGA. In Proceedings of the 1st European Conference on Artificial Life. The MIT Press, Cambridge, MA.
[19] M. Hataba, R. Elkhoully, and A. El-Mahdy. 2015. Diversified remote code execution using dynamic obfuscation of conditional branches. In Proceedings of the IEEE 35th International Conference on Distributed Computing Systems Workshops. 120–127.
[20] D. S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. Commun. ACM 18, 6 (June 1975), 341–343.
[21] M. L. Islam, N. Nurain, S. Shatabda, and M. S. Rahman. 2015. FGPGA: An efficient genetic approach for producing feasible graph partitions. In Proceedings of the International Conference on Networking Systems and Security (NSysS’15). 1–8. DOI: https://doi.org/10.1109/NSysS.2015.7043513
[22] P. Junod, J. Rinaldini, J. Wehrli, and J. Michelin. 2015. Obfuscator-LLVM—Software protection for the masses. In Proceedings of the IEEE/ACM 1st International Workshop on Software Protection. 3–9. DOI: https://doi.org/10.1109/SPRO.2015.10
[23] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In Proceedings of the Conference on Advances in Cryptology (CRYPTO’99). Michael Wiener (Ed.). Springer Berlin, 388–397.
[24] Xuejia Lai. 1992. On the Design and Security of Block Ciphers. Doctoral thesis. ETH Zurich. Retrieved from https://www.research-collection.ethz.ch/handle/20.500.11850/140723.
[25] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic optimization of program obfuscation. In Proceedings of the 39th International Conference on Software Engineering (ICSE’17). 221–231.
[26] Heiko Mantel and Artem Starostin. 2015. Transforming out timing leaks, more or less. In Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS’15). 447–467.
[27] Rabih Mohsen. 2016. Quantitative Measures for Code Obfuscation Security. Doctoral thesis. Imperial College London. Retrieved from https://spiral.imperial.ac.uk/handle/10044/1/55180.
[28] R. Mohsen and A. M. Pinto. 2015. Algorithmic information theory for obfuscation security. In Proceedings of the 12th International Joint Conference on e-Business and Telecommunications (ICETE’15). Vol. 04. 76–87.
[29] Rabih Mohsen and Alexandre Miranda Pinto. 2016. Evaluating obfuscation security: A quantitative approach. In Foundations and Practice of Security. Springer International Publishing, 174–192.
[30] Rabih Mohsen and Alexandre Miranda Pinto. 2016. Theoretical foundation for code obfuscation security: A Kolmogorov complexity approach. In E-Business and Telecommunications. Springer International Publishing, 245–269.
[31] V. Nandina, J. M. Luna, C. C. Lamb, G. L. Heileman, and C. T. Abdallah. 2014. Provisioning security and performance optimization for dynamic cloud environments. In Proceedings of the IEEE 7th International Conference on Cloud Computing. 979–981.

[32] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. 2017. HOP: Hardware makes obfuscation practical. In Proceedings of the Network and Distributed System Security Symposium (NDSS’17). The Internet Society, 1–15.

[33] M. Qiu, Z. Chen, J. Niu, Z. Zong, G. Quan, X. Qin, and L. T. Yang. 2015. Data allocation for hybrid memory with genetic algorithm. IEEE Trans. Emerg. Top. Comput. 3, 4 (Dec. 2015), 544–555. DOI: https://doi.org/10.1109/TETC.2015.2398824

[34] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In Proceedings of the 24th USENIX Security Symposium (USENIX Security’15). USENIX Association, Washington, D.C., 431–446.

[35] O. Reparaz, J. Balasch, and I. Verbauwhede. 2017. Dude, is my code constant time? In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE’17). 1697–1702.

[36] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21, 2 (Feb. 1978), 120–126.

[37] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Comput. Surv. 49, 1 (Apr. 2016). DOI: https://doi.org/10.1145/2886012

[38] Sebastiaan A. Terwijn, Leen Torenvliet, and Paul M. B. Vitanyi. 2011. Nonapproximability of the normalized information distance. J. Comput. Syst. Sci. 77, 4 (2011), 738–742.

[39] Meng Xu, Kangjie Lu, Taeso Kim, and Wenke Lee. 2017. Bunshin: Compositing security mechanisms through diversification. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC’17). USENIX Association, 271–283.

[40] W. Zeng and M. Y. Chow. 2012. Optimal tradeoff between performance and security in networked control systems based on coevolutionary algorithms. IEEE Trans. Industr. Electron. 59, 7 (July 2012), 3016–3025.

[41] W. Zeng and M. Y. Chow. 2013. Modeling and optimizing the performance-security tradeoff on D-NCS using the coevolutionary paradigm. IEEE Trans. Industr. Inf. 9, 1 (Feb. 2013), 394–402.

[42] Yan Zhuang. 2017. Methods for Improving the Quality of Software Obfuscation for Android Applications. Ph.D. Dissertation. Friedrich-Alexander-Universitat Erlangen-Nurnberg, Erlangen, Germany.

Received July 2020; revised October 2020; accepted December 2020.