Learning Without Peeking: Secure Multi-Party Computation Genetic Programming

Jinhan Kim[1], Michael G. Epitropakis[2], and Shin Yoo[1]

[1] School of Computing, KAIST, Daejeon, Republic of Korea
[2] Department of Management Science, Lancaster University, UK

Abstract. Genetic Programming is widely used to build predictive models for defect proneness or development efforts. The predictive modelling often depends on the use of sensitive data, related to past faults or internal resources, as training data. We envision a scenario in which revealing the training data constitutes a violation of privacy. To ensure organisational privacy in such a scneario, we propose SMCGP, a method that performs Genetic Programming as Secure Multiparty Computation. In SMCGP, one party uses GP to learn a model of training data provided by another party, without actually knowing each datapoint in the training data. We present an SMCGP approach based on the garbled circuit protocol, which is evaluated using two problem sets: a widely studied symbolic regression benchmark, and a GP-based fault localisation technique with real world fault data from Defects4J benchmark. The results suggest that SMCGP can be equally accurate as the normal GP, but the cost of keeping the training data hidden can be about three orders of magnitude slower execution.

1 Introduction

Genetic Programming is a variant of Genetic Algorithm that evolves programs and expressions instead of solutions [22]. While its recent popularity for Automated Program Repair (APR) [7,32] is closely related to GP's original ambition of automated programming, it has also been widely used by SBSE community to build predictive models for defect prediction [18], development effort prediction [6], and software quality estimation [17]. Recently, GP has also been successfully used to produce ranking models for fault localisation [12, 27].

While GP has been successfully applied to various problem domains, its application to each of the above domains requires access to potentially sensitive past data, such as historial defect pronenss data, information about internal resources and project cost, quality metrics, and test coverage data. GP uses the past data either to perform symbolic regression to find a model that fits the past results the best or to build ranking model that places the faulty program element as high in a ranking as possible.

The requirement on the use of sensitive past data raises a concern for both researchers and practitioners. It is difficult for researchers to study real world data, because data related to defects or internl resources can be regarded as highly

sensitive and may not be disclosed to external researchers. For practitioners, this rules out any form of Optimisation-as-a-Service type analysis. Hence we ask the following question: *is it possible to apply GP to learn predictive or ranking models for software engineering, without revealing sensitive data for training?*

This paper proposes a method that allows data to be hidden from GP, using Secure Multiparty Computation (SMC) [4], as an answer to our research question. SMC is a subdomain of cryptography whose goal is to enable multiple parties to jointly compute a function over their inputs while keeping the inputs hidden from each other. We instantiate Secure Multiparty Computation GP (SMCGP) using an SMC protocol called garbled circuits [35], and show that GP can be performed while not revealing the individual datapoints without loss of accuracy. We empirically evaluate the performance of SMCGP using a range of symbolic regression benchmark problems, as well as training of GP-based fault localisation model [27] using a real world fault data from Defects4J benchmark [11].

The technical contributions of this paper are as follows:

- We introduce the concept of SMCGP, the goal of which is to perform GP while remaining oblivious to the training data.
- We present an empirical study of well known symbolic regression benchmark problems, as well as a GP-based fault localisation technique in conjunction with the Defects4J repository. The results show that SMCGP is feasible without loss of accuracy, but requires significantly longer execution time.

Section 2 introduces Oblivious Transfer and Garbled Circuit, which is used to formulate SMCGP described in Section 3. Section 4 presents the experimental setup. Section 5 discusses the experimental results. Section 6 presents the threats to validity, and Section 7 contains the related work. Section 8 concludes.

2 Background

Secure Multiparty Computation (SMC) aims to perform distributed computation that involves multiple parties in a secure manner. In particular, its aim is to maintain each party's input to the computation process oblivious to other involved parties, while ensuring that the results are correct and uncorrupted.

Perhaps the most widely known example of SMC is the Yao's millionaires' problem, introduced by Andrew Yao [35]. Suppose there are two millionaires: both want to know who is richer without revealing the exact amount of one's wealth to the other. More formally, assume that there exist n participants, $p_1, \ldots, p_n$, each of which is holding private data, $d_1, \ldots, d_n$. SMC aims to compute the value of a public function over the set of all private data, i.e., $F(d_1, \ldots, d_n)$, while all participants keep their own data private.

Yao suggested the garbled circuit protocol, also known as Yao's protocol, to achieve secure computation between two parties (2PC). For more than two parties (MPC), secret sharing schemes such as Shamir Secret Sharing [26] are used. We formulate our GP under the 2PC context using garbled circuits, which is explained in the rest of this section.

2.1 Oblivious Transfer

In cryptography, oblivious transfer refers to a scenario in which the sender transfers one out of many possible messages without knowing what message has actually been transferred. Our choice of SMC, garbled circuits, is based on a specific type of oblivious transfer called 1-2 oblivious transfer [5]. Under the 1-2 oblivious transfer protocol, the sender has two strings, S_0 and S_1, and the receiver chooses $i \in \{0, 1\}$. After the transfer, the sender should not know which value of i the receiver chose, and the receiver should not know S_{1-i} (i.e., the string not chosen by the receiver).

The 1-2 oblivious transfer protocol can be implemented over asymmetric cryptography, such as the RSA [23]. The following is a brief description of Oblivious Transfer. Suppose Alice has two messages, m_0 and m_1, and Bob has a bit b. Bob wants to receive m_b without the sender knowing b. Let $N = pq$, where both p and q are large prime numbers; let e be relatively prime to $(p-1)(q-1)$. The encryption of message m is $m^e \mod N$. The transfer takes place as follows:

1. Alice generates an RSA key pair and sends the public exponent e to Bob. The private exponent, d, is secret.
2. Alice also generates and sends two random messages, x_0 and x_1, to Bob.
3. Bob chooses $b \in \{0, 1\}$, and generates a random k. Bob then sends $v = (x_b + k^e) \mod N$ (i.e., encryption of k blind to x_b) to Alice.
4. Alice computes $k_0 = (v - x_0)^d \mod N$ and $k_1 = (v - x_1)^d \mod N$. Alice knows k is one of these values, but does not know which.
5. Alice sends $m'_0 = m_0 + k_0$ and $m'_1 = m_1 + k_1$ to Bob.
6. Bob decrypts m'_b because Bob knows which x_b was chosen earlier.

Alice cannot determine which of x_0 and x_1 Bob chose. Bob cannot know the message he did not choose, as he can only unblind the message m_b with his k.

2.2 Garbled Circuit

The oblivious transfer deals with the secure transfer of messages: let us now turn to computation of functions for SMC. Garbled Circuit is a cryptographic protocol for two party secure computation. Intuitively, it operates by representing the function to be computed as a Boolean circuit and sending the circuit using the 1-2 oblivious transfer. We outline the process of garbled circuit transfer with a simple working example below:

1. Convert the function to be computed into a Boolean circuit with 2-input gates. As an working example, we are going to assume that our function itself is a logical AND. Table 1(a) shows the raw truth table.
2. Alice, the *garbler*, replaces 0s and 1s in the truth table with randomly generated string labels. The result is shown in Table 1(b).
3. Alice encrypts the output column(s) of the truth table with corresponding input labels. Alice also permutes the encrypted output rows so that the values cannot be guessed from the order (hence the name *garbled*).

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

(a)

a	b	c
X_0^a	X_0^b	X_0^c
X_0^a	X_1^b	X_0^c
X_1^a	X_0^b	X_0^c
X_1^a	X_1^b	X_1^c

(b)

Garbled Table
$Enc_{X_0^a, X_0^b}(X_0^c)$
$Enc_{X_0^a, X_1^b}(X_0^c)$
$Enc_{X_1^a, X_0^b}(X_0^c)$
$Enc_{X_1^a, X_1^b}(X_1^c)$

(c)

Table 1: Garbled Circuit Operation on $F(a, b) = AND(a, b)$: (a) the raw truth table, (b) Alice assigns random string labels to values in the truth table, (c) the output garbled table that is transferred.

4. Alice sends the encrypted circuit to Bob, along with her inputs. For example, if Alice's input for a is 1, Alice sends X_0^a. Since Alice generated the labels randomly, Bob does not know what Alice's actual input is.
5. In order to obtain the result, Bob needs the labels for his input. If Bob's input for b is 0, Bob asks for $b = 0$ between X_0^b and X_1^b through 1-2 oblivious transfer, after which Alice does not know which Bob chose between X_0^b and X_1^b and Bob does not know what the other label (in our case X_1^b) is.
6. Bob tries to decrypt each output row: he can only decrypt a single row, which is the output for the input from both Alice and Bob.

While our small working example only concerns a single logical operator as the function of interest, one can convert an arbitrary function into an optimised Boolean circuit [28] and apply the outlined process to the truth table of each 2-input gate within the circuit. By repeatedly applying the above process, Alice and Bob can securely compute the garbled circuit. The cost of privacy is the runtime overhead that stems from encryption and decryption as well as the conversion and execution of arbitrary functions as Boolean circuits.

2.3 Obliv-C

`Obliv-C` [37] is both a domain specific extension of C and a `gcc` wrapper that compiles the extension.[3] It is designed for developers to easily implement 2PC Secure Multiparty Computation: `Obliv-C` provides high-level interface to SMC via language extension, performs the Boolean circuit conversion, and handles the garbled circuit protocol. It has been applied to various privacy preserving machine learning scenarios [9, 29] as well as to email communications [10].

While we leave the low level implementation details of `Obliv-C` out in this paper (please refer to the original paper [37] for all the details), let us focus on two core language constructs, `obliv` qualifier and `obliv if` statement.

[3] It is available from `https://oblivc.org`.

– `obliv`: this qualifier denotes variables whose values need to remain oblivious. All oblivious variables are declared with the qualifier and assigned with actual values transferred from the garbled circuit protocol.
– `obliv if`: to prevent information leak from control flow, `Obliv-C` converts all control dependencies into data dependencies. This means that the body of `obliv if` will be always executed, regardless of how the branch predicate evaluates. When the predicate is false, the garbled circuit ensures that the values computed inside the block are simply ignored.

Figure 1 shows an example code of `Obliv-C` for the Yao's millionaires' problem. Variable `a` and `b` represent the wealth of two millionaires respectively. Using the function `feedOblivInt`, `a` and `b` are converted into an `obliv` qualified integers. The following if statement at Line 13 is a `obliv if` statement, because it makes a comparison between `obliv` qualified values. The result of comparison between `a` and `b` is stored in `result`, which is also `obliv` qualified variable. Finally, the call to `revealOblivBool` ensures that only the `result` is revealed to each party at the end of computation.

```
1  #include <million.h>
2  #include <obliv.oh>
3
4  void millionaire (void *args) {
5    ProtocolIO *io = args;
6    obliv int a, b;
7    obliv bool result = false;
8    a = feedOblivInt (io->myinput, 1);
9    b = feedOblivInt (io->myinput, 2);
10   obliv if (a < b) result = true;
11   revealOblivBool (&io->result, result, 0);
12 }
```

Figure 1: An `Obliv-C` program that implements Yao's Millionaires' Problem taken from Zahur et al. [37].

3 Secure Multiparty Computation GP using `Obliv-C`

In GP, the majority of the computation takes place during the fitness evaluation. In addition, this is the place where the training dataset is used by GP. This section describes how we can formulate SMC using the fitness evaluation as the function of interest. Our focus in this paper is the scenario in which multiple parties are holding different parts of the training dataset. We call this the multiparty dataholder scenario.

3.1 Multiparty Dataholder Scenario (2PC)

The multiparty dataholder scenario is a natural extension of the original Yao's millionaires' problem, as shown in Figure 2. We simply replace the function that returns the result of comparison between two numbers with the fitness

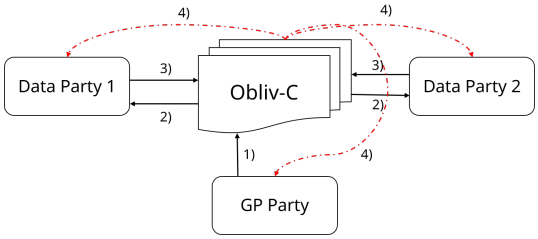

Figure 2: The multiparty dataholder scenario when there are two data parties and one GP party: 1) GP party generates the SMC program, and 2) sends it to each party. 3) Each party enters their input, and 4) the SMC program computes and all parties get the results.

function that evaluates the given GP candidate solution using the data held by the two participating parties. Let us call the data holders the *data parties*, and the mediator who is running the GP the *GP party*.

- **GP Party**: GP party executes the GP evolutionary loop, and generates `Obliv-C` based SMC program that contains the garbled circuits of the candidate solution to evaluate.[4] This SMC program is used by data parties to securely commit their inputs.
- **Data Party**: data parties hold the split training dataset. There are two ways a training dataset can be split. Suppose a training dataset contains n datapoints, each with m properties. A *horizontal* split means each data party holds mutually exclusive subset of the n datapoints (the union should be the entire training dataset). A *vertical* split means each data party holds mutually exclusive subset of the m properties of all n datapoints (the union of two property subsets should be the set of all m properties).

Whenever the GP party needs to evaluate a candidate solution, it first generates an `Obliv-C` source code that corresponds to the solution, builds it, and distributes the executable to data parties. Subsequently, data parties execute the SMC program and provide their parts of the split training dataset. Once all data parties enter their input, the fitness function computes and all data parties get the resulting fitness value. GP party receives the result and continues with the GP iteration until the predefined termination criterion is met. During the process, none of the data parties get to know more than their own shares of training dataset. Note that data parties do get to know what is being computed (i.e., which candidate solution the GP party is evaluating).

3.2 Singleparty Dataholder Scenario (1PC)

[4] In practice, our implementation gathers all candidate solutions in a generation and combines them all into a single `Obliv-C` program, to save the compilation overhead. This is similar to the approach taken by existing GPGPU based parallelisation approach for GP [14].

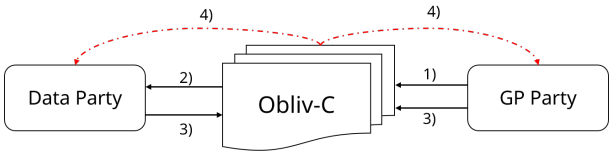

Figure 3: The singleparty dataholder scenario when there are one data party and one GP party: 1) GP party generates the SMC program, and 2) sends it to the data party. 3) The data party enters its input, whereas GP party enters nothing, and 4) the SMC program computes and all parties get the results.

As shown in Figure 3, we also present a singleparty dataholder scenario, in which the entire training dataset is held by a single participating party. We think this can also be a common use case for SMCGP, in which two stakeholders exist, one with the data (data party) and the other with Genetic Programming (GP party). The data party allows the GP party to learn from its data, but does not want to reveal the data. This scenario can be easily implemented by making the GP party to double as a data party with no training data subset to contribute.

4 Experimental Setup

This section presents out research questions, and describe experimental subjects and configurations.

4.1 Research Questions

This paper aims to compare our implementations of both single and multiparty data holder SMCGP to the Normal-GP through the following research questions.

- **RQ1. Effectiveness:** how well does the SMCGP perform compared to the Normal-GP?
- **RQ2. Efficiency:** what is the runtime overhead of SMCGP when compared to Normal-GP?

RQ1 is essentially a sanity check for `Obliv-C`: we should achieve the same level of effectiveness if `Obliv-C` performs oblivious and correct computation. **RQ1** is answered by comparing the Mean Squared Errors (MSEs) for the symbolic regression problems, and by comparing wasted effort (wef) for the GP-based fault localisation dataset: wef means the number of program elements which should be investigated before finding faulty program elements.

We use two-tailed Mann-Whitney U test to compare values from two different types of GP. The null hypothesis is that the mean values of different types of GP are the same. Failing to reject the null hypothesis would show that results from SMCGP cannot be distinguished from those of Normal-GP.

Our primary interest lies with **RQ2**, which investigates whether the runtime overhead of `Obliv-C` is practical. We expect both the use of garbled circuit

protocol and the communication overhead itself will have a negative impact on the execution time of SMCGP. Therefore, we answer **RQ2** by statistically comparing the execution time of Normal-GP and SMCGP.

4.2 Subjects

Subject	Equation	Size of Training Data	# of Variables
Keijzer-6 [13]	$\sum_i^x \frac{1}{i}$	50	1
Nguyen-7 [30]	$\ln(x+1) + \ln(x^2+1)$	20	1
Dow Chemical	Chemical Process Data	747	57
Vladislavleva-4 [31]	$\frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	1,024	5
FLUCCS [27]	Real-world Fault Data	7,280	40

Table 2: Four symbolic regression benchmark problems and one real world fault localisation data from Defects4J repository studied in this paper.

Table 2 shows the subjects of our experiment. We use four symbolic regression benchmark problems that have been widely studied in the literature [33], and one GP-based fault localisation technique and a real world fault dataset based on Defects4J repository [27].

Symbolic regression is a regresson analysis that aims to find a mathematical expression that best fits the given dataset [15]. Symbolic regression is usually performed by evolving trees that represent expressions, using the difference between the given data (i.e. the training data) and the data produced by candidate expressions as the fitness. Among the studied symbolic regression benchmark problems, Keijzer-6 [13], Nguyen-7 [30], and Vladislavleva-4 [31], are synthetic problems. On the other hand, the Dow Chemical symbolic regression dataset was the subject of the EvoCompetitions event at the 2010 edition of EvoStar conference and is based on real world industrial application at Dow Chemical.[5]

GP has been used for fault localisation to build ranking models: given various features (including data from both passing and failing test executions) for program elements as input, the aim is to learn a ranking model that places the faulty program element at the top. The expression evolved by GP returns what is called *suspisiousness score* for program elements, which are then sorted according to their scores.[6] For the fault localisation problem, we use the publicly available data from FLUCCS [27], which contains per-method Spectrum Based Fault Localisation (SBFL) scores [34], as well as various code and change metrics [27], for the faulty real world Java programs in the Defects4J benchmark [11].

[5] `http://dces.essex.ac.uk/research/evostar/competitions.html`

[6] Note that, while FLUCCS [27] makes a link between defect prediction and fault localisation via shared features, the GP formulations for two problems are different. Defect prediction *classifies* each program element to be fault prone or not: fault localisation *assigns suspiciousness scores* to program elements, aiming to place the faulty element at the top when ranked by them.

In our experiment, we select a single target program, `Mockito`, of which there exist 36 faulty versions in the FLUCCS dataset: each of the faulty version contains 1,040 methods on average. Out of 36 faulty versions, we use 32 for training, and use the remaining four for testing.

Since the 2PC scenario requires two data parties holding split dataset, we divide the original dataset in half. Datasets for the symbolic regression benchmark are split horizontally, whereas GP-based fault localisation dataset is split vertically (i.e., it results in generating two datasets that have 20 variables respectively). We posit that SMCGP will not be significantly slowed down for the 2PC scenario, as long as the network provides sufficient speed.

4.3 Configurations & Environments

We implement the GP party using DEAP [8], a Python library for evolutionary algorithm that includes an implementation of tree-GP. For fitness evaluation of each candidate GP tree, our GP party generates an `Obliv-C` source file using a template. To reduce the overhead of invoking `Obliv-C` compiler, we convert and compile the entire population in a single `Obliv-C` source file.

For symbolic regression benchmarks, we use a population size of of 40 individuals, a single point crossover with a rate of 0.6, and a subtree replacement mutation with a rate of 0.2. For the FLUCCS dataset, we use a population of 40 individuals, a single point crossover with the rate of 1.0, and a subtree replacement mutation with the rate of 0.1. Types of non-terminal GP nodes are addition, subtraction, multiplication, and safe division (i.e., $div(a, b) = \frac{a}{b}$ if $b \neq 0$ and 1 if $b = 0$). While parameter values may affect the quality of outcome, our main interest is the efficiency of SMCGP and not the solution quality.

We set the maximum tree depth to three and the stopping criterion to be after ten generations. While these may not be ideal choices for the accuracy, note that our primary aim in this empirical study is to investigate the impact of SMC on GP's efficiency and not to evolve the best possible solution. Note that, for the FLUCCS data, we do not use all 32 faulty versions simultaneously during training: rather, we randomly sample seven programs for every GP generation to lessen the burden of computation and mitigate overfitting.

We repeat each configuration of both types of GP 20 times. The experiments have been performed on machines equipped with Intel i7-6700 CPU and 32GB of RAM, running Ubuntu 14.04.5 LTS.

5 Results

Table 3 shows the results of Mann-Whitney U test on the MSE of SMCGP and Normal-GP, and Table 4 shows the results of Mann-Whitney U test on the wef value of SMCGP and Normal-GP. Based on these results, we conclude that there is *no statistically significant difference* between the results from SMCGP and Normal-GP, for both the symbolic regression benchmarks and the fault localisation problem ($\alpha = 0.05$). While this is as expected and should be, the sanity check through RQ1 was not wasted, as it enabled us to report a serious

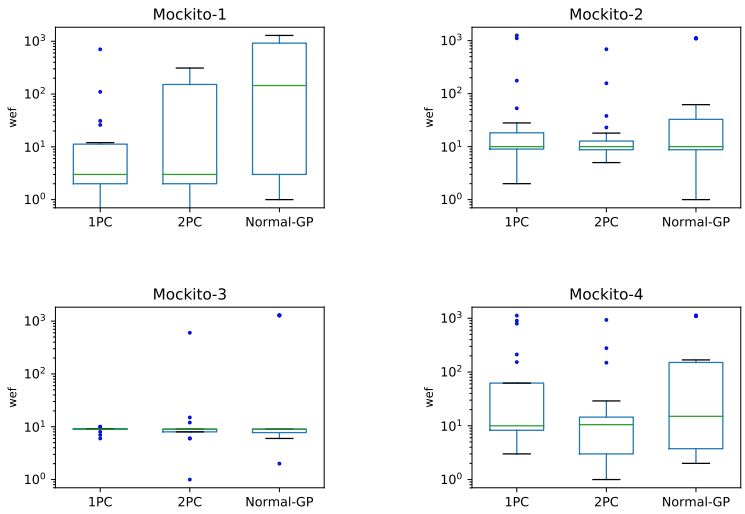

Figure 4: Boxplots of wef by each test program. The y-axis is shown on logarithmic scale.

defect in `Obliv-C`, which has been reported and subsequently patched by the developers of `Obliv-C`.

There is one exception, which is the p–values obtained from the case of Mockito-1. The p–values from 1PC and 2PC, 0.01 and 0.04 respectively, suggest statistically significant difference between SMCGP and Normal-GP. Figure 4 provides the possible reason for this: the 20 repeated runs of Normal-GP for Mockito-1 resulted in much higher average wef including more outliers. Both 1PC and 2PC SMCGP performed *better* than Normal-GP, hence the statistically significant difference. We attribute the poor performance of Normal-GP for Mockito-1 to two possible reasons: 1) stochastic nature of GP, regardless of whether the fitness evaluation is secure or not, and 2) the possibility that learning to localise the fault Mockito-1 is particularly challenging.[7]

In general, we conclude that the two samples of performance metrics from the studied problems are from the same distribution. We thereby answer **RQ1** as follows: there is no loss of accuracy in SMCGP when compared to Normal-GP.

For **RQ2**, we measure the execution time for each GP run. The results are shown in Figure 5. The differences in the execution time between 1PC and Normal-GP is significant: we observe that SMCGP is, on average, 1,739, 1,590, and 541 times slower than Normal-GP, for the Keijzer-6, Dow Chemical, and FLUCCS, respectively. The trend is similar between 2PC and Normal-GP. The main reason for this overhead is the use of garbled circuits protocol (i.e., generating and building `Obliv-C` SMC programs), as well as the TCP communication

[7] It is known that faults exhibit modal behaviours against fault localisation ranking models learnt by FLUCCS [27]: Mockito-1 may be one such a fault that can only be localised well by a small minority of ranking models.

Scenario	Subject	U–value	p–value
1PC, Normal-GP	Keijzer-6	220.5	**0.557**
	Nguyen-7	204.5	**0.910**
	Dow Chemical	204.5	**0.914**
	Vladislavleva-4	248.5	**0.183**
2PC, Normal-GP	Keijzer-6	156.5	**0.184**
	Nguyen-7	192.0	**0.833**
	Dow Chemical	179.0	**0.579**
	Vladislavleva-4	239.5	**0.272**

Table 3: The result of two-tailed Mann-Whitney U test on the MSE of SMCGP and Normal-GP. The significant level is 0.05 and the number of sample size is 20. The cases for which the p–values are not significant are typeset in **bold**.

Scenario	Test Program	U–value	p–value
1PC, Normal-GP	Mockito-1	105.5	0.010
	Mockito-2	205.5	**0.890**
	Mockito-3	210.0	**0.767**
	Mockito-4	205.0	**0.903**
2PC, Normal-GP	Mockito-1	124.5	0.040
	Mockito-2	194.5	**0.890**
	Mockito-3	182.0	**0.612**
	Mockito-4	161.5	**0.302**

Table 4: The results of two-tailed Mann-Whitney U test on the wef metric values from the FLUCCS dataset. The significance level is 0.05 and the number of sample size is 20.

(i.e., transferring the encrypted data), as the core GP configuration is the same for SMCGP and Normal-GP. Based on these observations, we answer **RQ2** as follows: the cost of data obliviousness in SMCGP can be up to three orders of magnitude slower execution time.

6 Threats to Validity

Threats to internal validity concern the extent to which the observed results from the empirical evaluation warrants our claims, such as implementation correctness. Both core components of our implementation of SMCGP, DEAP and `Obliv-C`, have been scrutinised as open source projects and widely applied to various work in the literature [9, 10, 14, 27]. The remaining parts of the implementation written by us have been carefully analysed manually to minimise the risk of implementation errors.

Threats to external validity concern the extent to which our empirical evaluation results generalise. We chose widely studied symbolic regression benchmarks as well as a real world SBSE application to promote generalisation.

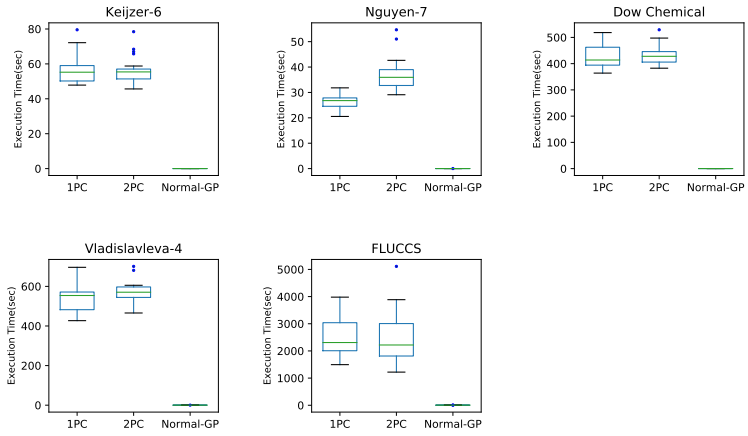

Figure 5: Boxplots of execution time by each subject.

Threats to construct validity concern how accurately the measurements we take are actually correlated to what they claim to measure. We assess the level of any threats to construct validity to be low, as our evaluation metric, MSE, is a standard evaluation metric for symbolic regression and based on actually observed errors.

7 Related Work

Genetic Programming evolves programs, often using trees as representation [22]. Its ability to evolve expressions rendered itself as a tool for predictive modelling in domains such as software development effort estimation [6] and defect proneness prediction [18]. It has been used to evolve risk evaluation formulas [36] as well as to learn more complicated ranking models [27] for fault localisation. Many application domain involve potentially sensitive data, which motivates our use of `Obliv-C` for SMCGP.

Peters et al. maintained the data privacy for cross-company defect prediction in which data from one company is used to train defect predictors for another [21]. The underlying technique is called MORPH: it obfuscates datapoints while ensuring that the obfuscated points do not cross the boundaries between the original and its neighbouring class. Li et al. later extended MORPH to Sparse Representation based Double Obfuscation (SRDO) with the same intention to preserve class labels [16]. Both techniques are designed for classification problems and need labels: SMCGP can be applied to any problems. Also, both techniques are much faster than SMCGP, they only obfuscate and not completely hide the data: SMCGP does not reveal any information.

There are other secure computation frameworks, both software and hardware based. Homomorphic Encryption (HE) allows computation on the encrypted data without the need to decrypt the data first [20], but is known to require inhibitively long execution time and significant memory usage. Hardware assisted

secure computation methods, such as Intel's Software Guard Extension (SGX) [1, 19], provide an enclaves in which user code can be securely executed. While SGX is an ideal solution for secure executions of a given specific program [3, 25], it does not support multiple parties and requires proprietary hardware.

Data privacy has been extensively studied in relation to machine learning [2, 24] but remains a relatively new topic for SBSE. As far as we know, ours is the first implementation of GP that attempts to completely hide training data while achieving the same computation.

8 Conclusion

We present SMCGP, a Genetic Programming that allows training data to remain private to data holders. We implement our version of SMCGP a Secure Multi-party Computation (SMC) protocol called garbled circuit, through a framework called `Obliv-C`. Our empirical evaluation of SMCGP using a set of widely studied symbolic benchmark and a fault localisation dataset from Defects4J repository shows that SMCGP is feasible without any loss of precision. However, the cost of hiding the data is about three orders of magnitude longer execution time. Future work will investigate the adversarial scenarios, in which the candidate solutions of GP also need to remain oblivious, as well as the possibility of application of secure multiparty computation model for other types of evolutionary algorithms.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (Grant No. NRF-2016R1C1B1011042).

References

1. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for cpu based attestation and sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13 (2013)
2. Balcan, M., Blum, A., Fine, S., Mansour, Y.: Distributed learning, communication complexity and privacy. In: COLT 2012 - The 25th Annual Conference on Learning Theory. pp. 26.1–26.22 (2012)
3. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp. 267–283. OSDI'14, USENIX Association, Berkeley, CA, USA (2014)
4. Du, W., Atallah, M.J.: Secure multi-party computation problems and their applications: a review and open problems. In: Proceedings of the 2001 workshop on New security paradigms. pp. 13–22. ACM (2001)
5. Even, S., Goldreich, O., Lempel, A.: A randomized protocol for signing contracts. Commun. ACM **28**(6), 637–647 (Jun 1985)
6. Ferrucci, F., Gravino, C., Oliveto, R., Sarro, F.: Genetic programming for effort estimation: an analysis of the impact of different fitness functions. In: Search Based Software Engineering (SSBSE), 2010 Second International Symposium on. pp. 89–98. IEEE (2010)

7. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation. pp. 947–954. GECCO '09, ACM (2009)
8. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research **13**, 2171–2175 (July 2012)
9. Gascón, A., Schoppmann, P., Balle, B., Raykova, M., Doerner, J., Zahur, S., Evans, D.: Privacy-preserving distributed linear regression on high-dimensional data. In: Proceedings on Privacy Enhancing Technologies. PPET 2017, vol. 4, pp. 345–364 (2017)
10. Gupta, T., Fingler, H., Alvisi, L., Walfish, M.: Pretzel: Email encryption and provider-supplied functions are compatible. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017. pp. 169–182 (2017)
11. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2610384.2628055
12. Kang, D., Sohn, J., Yoo, S.: Empirical evaluation of conditional operators in GP based fault localization. In: Genetic and Evolutionary Computation. pp. 1295–1302. GECCO 2017 (2017)
13. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) Proceedings of the 6th European Conference on Genetic Programming, Lecture Notes in Computer Science, vol. 2610, pp. 70–82. Springer Berlin Heidelberg (2003)
14. Kim, J., Kim, J., Yoo, S.: GPGPGPU: Evaluation of parallelisation of genetic programming using gpgpu. In: International Symposium on Search-Based Software Engineering, pp. 137–142. SSBSE 2017, Springer (2017)
15. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA (1992)
16. Li, Z., Jing, X.Y., Zhu, X., Zhang, H., Xu, B., Ying, S.: On the multiple sources and privacy preservation issues for heterogeneous defect prediction. IEEE Transactions on Software Engineering pp. 1–1 (2017)
17. Liu, Y., Khoshgoftaar, T.M.: Genetic programming model for software quality classification. In: Proceedings 6th International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking. pp. 127–136 (2001)
18. Maua, G., Galinac Grbac, T.: Co-evolutionary multi-population genetic programming for classification in software defect prediction. Appl. Soft Comput. **55**(C), 331–351 (Jun 2017)
19. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 10:1–10:1. HASP '13, ACM, New York, NY, USA (2013)
20. Moore, C., O'Neill, M., O'Sullivan, E., Doröz, Y., Sunar, B.: Practical homomorphic encryption: A survey. In: IEEE International Symposium on Circuits and Systems. pp. 2792–2795. ISCAS 2014 (June 2014)
21. Peters, F., Menzies, T., Gong, L., Zhang, H.: Balancing privacy and utility in cross-company defect prediction. IEEE Transactions on Software Engineering **39**(8), 1054–1068 (Aug 2013)

22. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com, available from http://www.gp-field-guide.org.uk (2008)
23. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (Feb 1978)
24. Sarwate, A.D., Chaudhuri, K.: Signal processing and machine learning with differential privacy: Algorithms and challenges for continuous data. IEEE Signal Processing **30**(5), 86–94 (Sept 2013)
25. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: Vc3: Trustworthy data analytics in the cloud using sgx. In: 2015 IEEE Symposium on Security and Privacy. pp. 38–54 (May 2015)
26. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (Nov 1979)
27. Sohn, J., Yoo, S.: FLUCCS: Using code and change metrics to improve fault localisation. In: Proceedings of the International Symposium on Software Testing and Analysis. pp. 273–283. ISSTA 2017, ACM (July 2017)
28. Songhori, E.M., Hussain, S.U., Sadeghi, A.R., Schneider, T., Koushanfar, F.: Tinygarble: Highly compressed and scalable sequential garbled circuits. In: IEEE Symposium on Security and Privacy. pp. 411–428. SSP 2015 (May 2015)
29. Tian, L., Jayaraman, B., Gu, Q., Evans, D.: Aggregating private sparse learning models using multi-party computation. In: NIPS Workshop on Private Multi-Party Machine Learning. PMPML 2016 (2016)
30. Uy, N.Q., Hoai, N.X., O'Neill, M., McKay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines **12**(2), 91–119 (Jun 2011)
31. Vladislavleva, E.J., Smits, G.F., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. IEEE Transactions on Evolutionary Computation **13**(2), 333–349 (April 2009)
32. Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st IEEE International Conference on Software Engineering. pp. 364–374. ICSE '09, IEEE (May 2009)
33. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better GP benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**(1), 3–29 (2013)
34. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707 (August 2016)
35. Yao, A.C.C.: How to generate and exchange secrets. In: Proceedings of the 27th Annual Symposium on Foundations of Computer Science. pp. 162–167. SFCS '86, IEEE Computer Society, Washington, DC, USA (1986)
36. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In: Fraser, G., Teixeira de Souza, J. (eds.) Search Based Software Engineering, Lecture Notes in Computer Science, vol. 7515, pp. 244–258. Springer (2012)
37. Zahur, S., Evans, D.: Obliv-c: A language for extensible data-oblivious computation. IACR Cryptology ePrint Archive **2015**, 1153 (2015)