Subdomain and Access Pattern Privacy
Trading off Confidentiality and Performance

Johannes Schneider[1], Bin Lu[2], Thomas Locher[1], Yvonne-Anne Pignolet[1],
Matus Harvan[1] and Sebastian Obermeier[1]
[1]*ABB Corporate Research, Baden-Daettwil, Switzerland*
[2]*Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland*

Keywords: Secure Computing, Remote Data Processing, Privacy-preserving Cloud Computing, Security Engineering, Industrial Systems.

Abstract: Homomorphic encryption and secure multi-party computation enable computations on encrypted data. However, both techniques suffer from a large performance overhead. While advances in algorithms might reduce the overhead, we show that achieving perfect (or even computational) confidentiality is not possible without increasing the running time compared to computations on plaintext more than exponentially in some cases. In practice, however, perfect confidentiality is not always required. The paper discusses mechanisms to trade off confidentiality and performance for computing on ciphertexts. It introduces a fine-grained approach to define security levels for variables called *(statistical) subdomain privacy*. This concept differs substantially from prior work because it treats a variable as confidential or non-confidential depending on the actual value. We further propose privacy-preserving methods for memory access patterns. We apply our techniques to improve performance of control flow logic (loops, if-then-else logic) and arithmetic operations such as multiplications. The evaluation shows that the resulting speedup can be in the order of several magnitudes depending on the privacy needs.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Trusting a third party to handle confidential data can be an issue, especially if the third party cannot be well audited. Modern cryptography makes it possible to encrypt data in a manner such that meaningful computations can be carried out directly on the ciphertext, i.e., without revealing confidential information to a third party. In principle, this enables cloud computing services that work on private data without violating confidentiality. In the industrial sector, for example, customers do not have to reveal their private data but could still leverage analytical services carried out in an (untrusted) cloud. The key obstacle that has hindered the widespread availability of such services is the inherent performance bottleneck of current techniques enabling computations on ciphertexts such as fully homomorphic encryption (FHE) or secure multiparty computation (SMC). Although a lot of algorithmic improvements have been made in the last years, there are inherent limitations in terms of performance of complex programs running on encrypted data that cannot be resolved without lowering the requirements on data confidentiality. This fundamental issue has not been addressed yet as prior work either considers restricted programs or tolerates large performance penalties, rendering secure computation impractical in many cases. For example, executing both branches of nested "if" statements to hide the value of the corresponding condition can lead to a drastic increase in running time to avoid leakage of the value of the condition and, in turn, of the confidential data on which the condition depends.

In this paper, we introduce the concept of *subdomain privacy*—a new approach to defining privacy depending on the value of a variable. Thus, the value of a variable is not necessarily kept secret for the whole execution of the program, but might be revealed depending on the concrete value it holds at runtime. One could also speak of *static* and *dynamic* privacy. We show how this mechanism facilitates trading off security and performance depending on the scenario. For "if" statements, our approach can imply executing (with some probability) either both branches or just one branch, leading to significant performance gains. In particular, we leverage the observation that

Schneider, J., Lu, B., Locher, T., Pignolet, Y-A., Harvan, M. and Obermeier, S.
Subdomain and Access Pattern Privacy - Trading off Confidentiality and Performance.
DOI: 10.5220/0005954100490060
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 49-60
ISBN: 978-989-758-196-0
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

for a variable holding a value of some (large) domain of values, the (needed) confidentiality often depends on the actual value of the variable. Rather than declaring the variable to be confidential for its entire domain of values, only a certain set of values is treated as confidential. As a result, we can even disclose the value of the variable if it is not part of the confidential set of values. Subdomain privacy enables the system to select algorithms at run-time that work on non-confidential data and thus are significantly faster than their counterparts operating on encrypted data only.

For example, an industrial plant hosts control devices that operate on sensor data. Optimization algorithms analyze control actions based on sensor values for immediate control but also for long-term improvement. In a typical scenario, sensor values in a normal range often do not need to be kept confidential since this information and the corresponding control operations might be considered common knowledge. However, the control actions and sensor data dealing with abnormal cases is often based on long-term experience of the control system vendor that is hard to obtain, and thus should be kept confidential.

Another example are medical records for rare diseases. The privacy of patients suffering from a disease should be protected and their medical records should be kept confidential. Examination data of healthy patients should in principle be kept confidential as well but it is generally much less sensitive. Thus, rather than not ensuring any confidentiality at all, one might selectively encrypt or decrypt variables depending on the actual content of a variable. For example, we might process all data of sick persons in an encrypted manner and all others as plaintexts. However, it might also be valuable to operate only on some healthy patients in a non-encrypted manner. For example, assume that we investigate upon 100 health records of people from a mid-sized company. If there is one sick person then just one record would be kept encrypted and all other 99 records are unencrypted. Thus, given the complete list of 100 people and those 99 that are treated as plaintexts, one will easily identify the one sick person, although her health record is encrypted.

In this work, we formalize these ideas and concerns and provide a multitude of examples for which the developed concepts are applicable and also demonstrate how these concepts can be used. We also give an evaluation using a recent multi-party scheme showing that, depending on the scenario, performance improvements of several orders of magnitude are possible.

1.1 Contributions

In short, the *contributions* of this work are the following. We introduce the notion of sub-domain privacy, a new concept for trading off confidentiality and performance. Moreover, we discuss practically relevant applications of this concept. In addition, we propose methods to prevent an attacker from exploiting memory access patterns to derive insights about the data and computation. Finally, we experimentally evaluate our concepts on selected applications.

1.2 Outline

In Section 2, we describe the overall setup including a threat model. Impossibility results showing that achieving perfect (or computational) security might prove infeasible due to a theoretically unbounded increase in running time are presented in Section 3. Section 4 introduces the concept of subdomain privacy and Section 5 dicusses applications thereof. Our approach to achieve access pattern privacy is presented in Section 6. Section 7 provides an evaluation of these concepts. Related work is discussed in Section 8 and Section 9 concludes the paper.

2 THREAT AND COMPUTATIONAL MODEL

We consider a client-server setting. The client encrypts data and transmits encrypted data to a server in case of homomorphic encryption. In case of secure multi-party computation (SMC), the client transmits keys and encrypted values to a set of servers, such that no server on its own can decrypt any ciphertext. Although the client might be involved in the computation, the client should perform as few operations as possible on the data aside from the initial encryption. We consider passive attackers that cannot alter any values but they can monitor computations, memory (values of memory cells and access patterns), and data stored (at the third party). Furthermore, the attackers can measure the execution time of the computations. We assume that the cryptographic schemes used, are strong enough so that an attacker cannot break the encryption (for reasonable key lengths) without access to the decryption key. For SMC we assume that the system is secure as long as no two parties collude. We also assume that an attacker has knowledge about input distributions to pograms, ie. for a function $f(x)$ that should be evaluated on encrypted inputs x the attacker knows the distribution of the plaintexts used to call function f.

3 IMPOSSIBILITY RESULTS

In this section, we provide general statements with respect to the overhead that comes with keeping input and output confidential.

Computation on encrypted data (performed by some program) achieves *perfect security (confidentiality)* if an attacker with capabilities as described in Section 2 cannot guess the plaintext value of the encrypted input and output values of the program with a probability higher than someone could guess by observing just the encrypted input and output values. Thus, the attacker gains no additional knowledge about the encrypted data through the execution of the program. The following theorem states that guaranteeing perfect secrecy can be costly.

Theorem 1. *For some deterministic programs, any possible trace (branch) of the program must be executed in order to achieve perfect security.*

Proof. Consider a program for which every "if" condition depends on a secret input value. Assume that an "if" condition is evaluated and only one of the two possible branches is executed. An attacker observing that a branch is not executed gets to know the outcome of the condition, which in turn leaks information upon the input. $\square$

A trace for a specific input manifests itself also in its memory access pattern. Thus, Theorem 1 also applies to memory locations.

Corollary 1. *For some programs, all memory locations that could be accessed by one of the feasible inputs must be accessed for any given input in order to achieve perfect security.*

If there is an n-fold nested "if" statement, Theorem 1 says that potentially all 2^n branches may have to be executed. Naturally, executing all branches can lead to an exponential increase of the running time. However, the running time may even increase much more in the worst case.

Theorem 2. *There are deterministic programs that must run at least as long as it takes to execute the input that maximizes its running time in order to achieve perfect security.*

Proof. Let the maximal time of the program be t_{max} and assume that it occurs for input values I. If the execution time is less than t_{max}, the attacker can deduce that the input was distinct from I. $\square$

Since the running time for some inputs may be arbitrarily larger than for the actual input(s), the running time of a program guaranteeing perfect security can be arbitrarily larger than the running time of the unsecured version.

4 SUBDOMAIN PRIVACY

A variable is typically seen as either confidential or non-confidential, independent of the actual value it stores. A variable v might store a value $val(v) \in D$ of a large domain D. For instance, a 32-bit integer variable v has $val(v) \in [-(2^{31}), 2^{31}-1]$. In many application scenarios revealing the value of a variable or parts of it might be tolerable for some values of the domain, but not for all. For temperature measurements, it might be tolerable to disclose a measured value if the temperature is within a certain "normal" range say [0,100] degrees but not if it is outside of this range. To capture such characteristics, we introduce the general notion of subdomain privacy. It enables the refinement of a variable's confidentiality requirements. Refining security constraints makes it possible to employ fast algorithms operating on non-encrypted data for certain values (or parts of a value).

The examples above lead to a conditional confidentiality definition, where confidentiality is only ensured if a value is in a certain set of values. More formally:

Definition 1 (Subdomain Privacy (with respect to C))**.** *A variable v with $val(v) \in D$ is subdomain private with respect to a set of values $C \subseteq D$ if given that $val(v) \in C$ an attacker cannot do better than guessing $val(v)$ with probability $1/|C|$. We write $DOM(v,C)$.*

The definition implies that $val(v)$ is perfectly secure for $C = D$. One might also say that the variable v is perfectly secure with respect to C, meaning that we reveal whether $val(v)$ is in C or not but nothing else about $val(v)$ if the value lies in C. In practical terms this means that given an encrypted variable $v \in D$, e.g. as input, we can (often) check at the beginning of the program if $val(v) \in C$. If so, we must keep the value of v encrypted, otherwise computations can be done using its plaintext. The set C can itself be kept secret. Under the assumption that over time a variable v takes all values in D, keeping C encrypted adds no value, since C can be inferred as being all values that have not been observed in plaintext.

The concept of subdomain privacy must be applied with great care, since it might lead to information leakage of confidential variables. The first problem arises if in the beginning it seems that we can reveal v because $val(v) \notin C$, but later v is assigned a value in C. More precisely, say in the beginning $v := x_0 \notin C$. At some later point in time, v is assigned the value $x_1 \in C$. If there is any dependency between x_0 and x_1, i.e., $prob(v = x_1 \mid v = x_0 \text{ before}) \neq prob(v = x_1)$, then we must keep v encrypted throughout the computation. A concrete example where we have this dependency would be $if(v = x_0)\ v := x_1$. This principle of dependency (or confidentiality propagation) also extends to any other variables w and their values on which the assignment to v might depend. If a variable v is read only, this problem does not arise.

It may often not be possible to disclose whether $v \notin D$ (with certainty). For instance, assume a system reveals data of all healthy patients of a hospital as plaintext for research purposes, ie. these could be patients coming for routine checkups. It keeps all other patient information confidential. Then, knowing that someone is a patient and whether or not its data is encrypted, makes it possible to determine whether or not this person is healthy. In order to at least ensure statistical security, we can disclose whether a person is healthy, i.e. $val(v) \notin D$, only with some probability. In our example, we could reveal the data of a healthy patient with a 90% rather than a 100% probability. Therefore, the information of all sick patients as well as 10% of all healthy patients are treated confidentially. Assume that 0.1% of patients have a rare disease. Then, knowing that the information of a person is treated confidentially, still ensures some degree of statistical security. As we shall see later in detail, if a person is treated confidentially there is only about 1% chance that this person is actually having the rare disease. An attacker cannot gain more certainty about the health status given all underlying primitives are perfectly secure.

Therefore, as before we want to conceal what value a variable stores given that this value is from some set of values. In addition, we also do not want to reveal with certainty whether the variable has any of the values of this set or not. We provide statistical security for the information whether a value v is in C. *Statistical subdomain privacy* is formally defined as follows.

Definition 2 (Statistical Subdomain Privacy (with respect to membership in C))**.** *A variable v with assigned fixed value $val(v) \in D$ is statistically subdomain private with respect to membership in C if v is subdomain private with respect to C and we reveal a $val(v) \notin C$ with probability at most $p_S \in [0,1]$. We write $STATDOM(v,C,p_S)$.*

Note by definition, we have that subdomain privacy equals subdomain privacy, if we reveal the result $v \notin C$ with probability one, i.e. $STATDOM(v,C,1) = DOM(v,C)$. If we never reveal whether $v \notin C$, i.e. we use $STATDOM(v,C,0)$ then v must always be treated confidentially, even for a value that is not in C. The probability that an attacker can guess whether $v \in C$ given that it observed that a value v is treated confidentially, depends on the probability distribution of values $val(v) \in D$, in particular, $prob(val(v) \in C)$. In the simplest case, if all values have to be treated confidentially, i.e. $prob(val(v) \in C)$, an attacker can trivially guess whether $val(v) \in C$. Using statistical subdomain privacy in addition to the fraction the probability that a value is encrypted is given by $prob(val(v) \in C) + prob(val(v) \notin C)) \cdot (1 - p_S)$. Thus, an attacker can guess $prob(val(v) \in C)$ for $STATDOM(v,C,p_S)$ with at most the following probability:

$$p(\text{Guess } val(v) \in C) \leq \frac{prob(val(v) \in C)}{prob(val(v) \in C) + prob(val(v) \notin C) \cdot (1 - p_S)}$$

For illustration, let us look at the previous example. Assume that 0.1% of all persons have a rare disease, ie. probability $prob(val(v) \in C) = 0.001$ and we disclose whether a person is healthy with 90%, ie. probability $p_S = 0.9$. In this case, an attacker can guess whether a variable that remains encrypted holds data of a person with a rare disease with only about 1%, ie. $p(\text{Guess } val(v) \in C) = 0.0099$. From a practical perspective the question arises what happens, if a variable v is assigned different values throughout the execution of a program. For instance, for $v := x$ and later $v := y$, should we ensure that one cannot guess whether $x \in C$ or $y \in C$ with probability p, i.e. $prob(\text{Guess } x \in C \vee y \in C) \leq p$ or is it sufficient to ensure that no individual assignment can be guessed with probability p, i.e. $prob(\text{Guess } x \in C) \leq p$ and $prob(\text{Guess } y \in C) \leq p$? Clearly, the first condition provides better security. However, it might also limit the usefulness of statistical subdomain privacy, i.e. assume we have $prob(\text{Guess } x \in C) = p$ then we have essentially reached the maximum amount of information to be disclosed for the variable, thus starting from the second assignment we must always hide the fact whether the value is in the confidential set. The above definition of subdomain privacy, only focuses on the case, when the variable is assigned once. We advocate the scenario that after every assignment to v we might leak its value with probability p .

Next, we discuss how to check for membership $val(v) \in C$ and present an efficient way to define sets C based on ranges. Subsequently, we show in more detail what computations are necessary to ensure statistical security.

4.1 Range Partitioning

In principle, the set of values $C \subseteq D$ for which confidentiality has to be ensured can be defined arbitrarily. However, computing whether $val(v) \in C$ can introduce significant overhead depending on the structure of C. Therefore, we focus on defining C based on continuous ranges that allow for efficient membership computation $val(v) \in C$. We assume that an ordering is defined on the domain D, i.e., for $a, b \in D$ we can compute $a \leq b$. Two ways to define partitionings are:

- Domain range partitioning: Let a subdomain range C_r be a (continuous) range of values $C_r = [c_0, c_1]$ with $c_0, c_1 \in D$.
- Value range partitioning: Consider a value $a \in D$. Let $a = (a_{n-1}|a_{n-2}|...|a_0)$ be the splitting of a into n parts. A value range partitioning $C_v = [c_0, c_1]$ based on a part j is given by all values $a \in D$ such that for part j holds $a_j \in C_v$.

If a and b are available in encrypted form, deciding whether a value is in the given range requires the ability to carry out comparison operations. This can be achieved with MPC protocols or by involving the client. For domain range decisions typically two comparisons with ciphertexts are necessary, i.e., one with the lower and one with the upper bound of the range. It requires only one comparison if the lower (or upper bound) is the minimum (or maximum) element of the domain. Checking whether a value is part of a value range C_v is more involved, since it requires more operations in addition to the two comparisons to extract the part of a value that is relevant for membership computation. Depending on the encryption scheme, one shift and one modulo operation can be sufficient. Note that value range partitioning is a generalization of domain range partitioning. The membership computation $val(v) \in C$ either yields the result in an non-encrypted manner, i.e., a boolean *res* that stores whether $val(v) \in C$ or an encrypted boolean $ENC_K(res)$. Some multi-party protocols are inherently designed for the scenario where the result of a secure computation should be disclosed to the participants of the computations. Generally, having the requirement that a server should obtain the result of some computation in plaintext puts confidentiality at risk. For example, if a server has some (limited) capability to gain information on potentially confidential variables, it might abuse this capability by issuing non-legitimate, additional or modified queries. In the semi-honest model this does not pose any security risk because computations are performed faithfully. For stronger adversarial models, one way to deal with this using FHE is to involve the client, which verifies whether a request for revealing a value is legitimate. One can also add more parties to split responsibilities, i.e., mixing FHE with SMC. For example, one party might perform homomorphic computations and another party might only decrypt some values. For SMC it must be ensured that no single party can violate subdomain security. More precisely, no single party should be able to trick the other parties into believing that they should reveal a value $val(v)$ even though it should be kept confidential, i.e., it actually holds that $val(v) \in C$. One (general) method is that parties must share their (partial) outcomes synchronously to be able to decrypt the result, e.g., using commitments. For illustration, consider two parties A, B, A having a key K and B having an encrypted value $res + K$. The parties want to share $res \in \{0, 1\}$. If party A sends K to B, then B can obtain *res*. B can make A believe that *res* is an arbitrary value by transmitting $res + K + x$ with $x \in \{-1, 0, 1\}$. One solution involves two communication rounds: In the first round, both parties exchange their encrypted results together with hashes of the keys. Once both parties have received the message from the other party they exchange the keys. A party commits to its key in the first round, i.e., it cannot change its choice after having received an encrypted value (and key) from any other party.

4.2 Statistical Security

We ensure statistical security by revealing $val(v)$ only for a fraction of all values $val(v) \in (D \setminus C)$ that do not have to be confidential. More precisely, we reveal $val(v)$ with probability p, ie. we use a random Bernoulli variable $b \in \{0, 1\}$, that is one with probability p and zero with probability $1 - p$. Computing a ciphertext of such a variable b without client involvement can be done in a multi-party setting: In order for b to be one with probability p, we first compute a random number $r \in [1, n]$ (for some large value n). Then b is one if $r \leq n \cdot p$. More precisely, each party P_i picks two large random numbers $r_i, k_i \in [1, n]$. They exchange $r_i + k_i \mod n$. Once each party has received these values from all other parties, they compute the encrypted value of $u = \sum_i r_i \mod n$, i.e., $ENC(u) = \sum_i (r_i + k_i)$. The encrypted bit b is then obtained using a secure comparison, ie. $ENC(b) = ENC(u \leq p \cdot n)$, encoded as 1 if the inequality holds and 0 otherwise.[1] For statistical security, b is multiplied with the encrypted result $res \in \{0, 1\}$ of $val(v) \notin C$, to get $b \cdot res$, which is then decrypted. This scheme causes $val(v)$ to be revealed whenever b is 0. If b is 1 then the value is revealed only if the value is not among the set of

[1]Instead of this procedure, precomputed randomness could be used.

values for which $val(v)$ has to be kept confidential. Without statistical security a value is always revealed when the result *res* from checking whether $val(v) \in C$ is true. Thus, we reveal fewer values in expectation with statistical security than without statistical security.

For FHE it is possible that a server runs a pseudorandom number generator where the seed and all generated random numbers are encrypted. However, this is not sufficient, since a decision must be made based on the chosen random number. If the server can decrypt the random number and thereby trigger a decryption of a potentially confidential value, we have the same problem as discussed for domain range partitioning, i.e., a server that is not acting according to the honest-but-curious model might abuse its power to gain confidential information. As mentioned above, additional servers or communication with the client can be used to prevent this.

5 APPLICATIONS

Next, we discuss a variety of applications of subdomain security and statistical subdomain privacy.

5.1 Secure Operations

The definition of subdomain privacy can be applied in a straightforward manner to speed up many operations. Consider a domain range partitioning $C_r = [c_0, c_1]$. Assuming that a value $val(v) \notin C_R$ is available in plaintext makes many operations significantly faster. This may be obvious if all operands of an operation are in plaintext, but it also holds for a variety of operations even if only one operand is in plaintext and the other remains encrypted.

The multiplication of a ciphertext and a plaintext can be carried out efficiently with additive encryption schemes. E.g., in a multi-party setting (such as (Schneider, 2016)) computing $ENC(a \cdot b)$ given a in plaintext and $ENC_K(b)$ encrypted with key K, where one party holds $ENC(b) = b + K$, another party K, the parties can both multiply their values by a, i.e. one party gets $a \cdot ENC(b) = ab + aK$ and the other aK. Decryption of $a \cdot ENC(b)$ with aK indeed yields the product, i.e. $a \cdot ENC(b) - aK = a \cdot b$. This multiplication can be done without communication between the two parties. In contrast, the same operation requires several message exchanges if both a and b are encrypted. Similarly, the additively homomorphic Paillier cryptosystem (Paillier, 1999) supports the multiplication of a ciphertext and a plaintext, while it is not possible to multiply two ciphertexts. Therefore, subdomain privacy can in some cases enable the use of the Paillier cryptosystem rather than resorting to fully or somewhat homomorphic encryption mechanisms, which are typically much slower.

Value range partitioning is generally less effective than domain range partitioning and it often requires more care. For each value $val(v)$ a plaintext part and an encrypted part needs to be maintained. Associative operations can then be carried out using different algorithms for each part. As an example, assume that the last k bits of a variable are not confidential. Thus, we have for a value $val(v)$ an encrypted part enc_v for the higher order bits and a plaintext part $plain_v$ for the last k bits. Multiplying two variables u, v requires computing $val(u) \cdot val(v)$ in the following way. We compute the part of the product $u \cdot v$ that must remain encrypted, i.e., $enc_{u \cdot v} = enc_u \cdot enc_v \cdot 2^{2k} + (enc_u \cdot plain_v + enc_v \cdot plain_u) \cdot 2^k$ and the part of the product that is kept as plaintext, i.e., $plain_{u \cdot v} = plain_u \cdot plain_v$. Even though this increases the number of operations, the duration of the multiplication where both operands are ciphertexts typically exceeds the others considerably. The running time of ciphertext operations typically increases at least linearly with the (bit) size of a value. Thus, reducing the number of bits of an operand might yield computational savings for an overall performance improvement. Note, that $plain_{u \cdot v}$ might be larger than 2^k and therefore an attacker might gain some information about bits other than the last k bits. Depending on the application this might be acceptable or not. If this is not allowed, one can use only $k/2$ bits per plaintext part, which ensures that nothing is leaked for an individual operation. For multiple operations, it can be necessary to further reduce the size of the plaintext parts.

Another operation that benefits from value range partitioning is finding an encrypted value in a sorted array. Comparisons are generally expensive operations when performed on encrypted data. In case, we do not have to keep the last k bits for numbers consisting of n bits confidential, we can organize the data structure such that we first search for prefixes of the first $n - k$ bits. For a value a to be searched we find the largest value that is smaller than the prefix of a and then the smallest value that is larger than the prefix of a. The exact value can then be found by looking only at those values in the range for which the last k bits match. For binary search this can reduce the number of comparison on encrypted data from $\log n$ to $\log(n/k)$. Binary search might leak access patterns (for non-oblivious data access), which not be acceptable. Resorting to linear search for hiding access patterns could solve this problem. Subdomain privacy reduces the comparisons on secure data from n to n/k.

5.2 Secure Conditions

Securing conditions requires executing both branches of a condition. This can lead to a prohibitive increase in running time according to Theorem 2. Using (statistical) subdomain privacy, the evaluation of some conditions can be disclosed without violating confidentiality constraints and thus only one branch needs to be executed. In other words, confidentiality can be maintained by revealing the evaluated condition in some cases, depending upon the variables and their values involved in the evaluation of the condition. This enables the system to only execute one of the two branches in these cases.

Our notions of (statistical) subdomain privacy covers the scenario where we want to secure a statement of the form:

$$\text{if(cond) then } CodeBlock_A; \text{ else } CodeBlock_B;$$

One straightforward way is to disclose the evaluation of the "if" condition with some probability p irrespective of the values involved, e.g., by defining a variable $STATDOM(v, \{true, false\}, p)$ with $v := cond$. A slightly more advanced option is to reveal the result of the condition only in case it is true (or false) with some probability, i.e., by defining $STATCOM(v, true, p)$. The second option is of particular interest if one branch is rarely executed but is very costly to execute. In some cases, with subdomain privacy, conditions can be evaluated on non-encrypted data. In the example from the previous section, where the last k digits are not kept confidential for a variable v, a condition

$$\text{if}(a \mod 2^k = 0)$$

can be evaluated using plaintext operations.

When using a multitude of nested "if" statements, several strategies can be employed to balance security and performance, e.g., following all traces until the number of traces has reached a certain limit, selectively revealing the result of "if" conditions depending on their information leakage, or branching factor or making the choice dependent on the runtime overhead of a branch. These strategies can be implemented using the concept of subdomain privacy.

5.3 Secure Loops

In order to perfectly secure a loop we must execute the maximal number of possible iterations for any input according to Theorem 1. This can cause an extremely high overhead compared to executing only the operations necessary to fulfill an algorithm's purpose. Therefore, we often have to balance security and performance. A simple approach is to limit the maximal number of iterations to a fixed value. If any more iterations are needed, then the leakage of information must be either implicitly accepted or variants of the loops that are performed on non-confidential data have to be carried out. Another approach is to use statistical subdomain privacy and reveal the evaluation of a condition only with a certain probability p after each iteration. Thus, in the example (see Algorithm 1) we can define a condition variable $cond := F_0(X) \in \{0,1\}$ and define $subCond := STATDOM(cond, \{\}, p)$. This means that we can reveal any value of $cond$ (i.e., $\{0,1\}$), but only do so with probability p. In Algorithm 1 we use the function $Revealed(subCond)$ to indicate whether subCond can actually be revealed in this iteration or not. One might also choose a deterministic function depending on the current iteration number to decide when to reveal a value.

Algorithm 1: Securing a while loop.

```
1:  {Original Loop with Variables $X$ and Functions $F_0, F_1$}
2:  **WHILE**($F_0(X) = 1$) $X = F_1(X)$
3:
4:  {Transformed, Perfectly Secure Loop}
5:  $maxIter :=$ Maximum iterations for any input $X$
6:  $n_i := 0$
7:  **repeat**
8:      $ENC(cond) := ENC(F_0(X))$
9:      $ENC(X) := ENC(cond) \cdot ENC(X) + (1 - ENC(cond)) \cdot ENC(F_1(ENC(X)))$
10:     $n_i := n_i + 1$
11: **until** $n_i = maxIter$
12:
13: {Secure Loop With Subdomain Private Condition}
14: **repeat**
15:     $ENC(cond) := ENC(F_0(X))$
16:     $ENC(X) := ENC(cond) \cdot ENC(X) + (1 - ENC(cond))ENC(F_1(ENC(X)))$
17:     $subCond := STATDOM(F_0(X), \{\}, p)$ {Evaluate Loop-Condition and reveal it with probability $p$}
18: **until** $Revealed(subCond)$ AND $subCond = 1$
19:
20: $n_i := 0$ {Secure Loop With Fixed Intervals}
21: $nextReveal := c_0$ {we perform $c_0$ iterations until first check}
22: **repeat**
23:     **repeat**
24:         $ENC(cond) := ENC(F_0(X))$
25:         $ENC(X) := ENC(cond) \cdot ENC(X) + (1 - ENC(cond)) \cdot ENC(F_1(ENC(X)))$
26:         $n_i := n_i + 1$
27:     **until** $n_i = nextReveal$
28:     $nextReveal := nextReveal \cdot c_1$ {$c_1$ is increment factor}
29: **until** $F_0(X) = 1$
```

So far we have used subdomain privacy to reveal the evaluation of a condition with a certain probability p. This means that the evaluation of the condition is revealed on average every $1/p$ iterations. Here, we present an alternative that is deterministic and reveals a condition depending on the number of current iterations. This makes it easy to ensure certain degrees of privacy as well as runtime guarantees. For example, by always doubling the number of iterations until the evaluation of a condition is revealed we can ensure that the number of iterations is at most twice as much as the actual (minimum) number of needed iterations. In Algorithm 1, a minimum number of iterations c_0 is always executed without disclosing the evaluation of any condition. Then, the total number of iterations that are performed until the next condition is evaluated and disclosed grows by a factor of c_1 each time the condition is evaluated. This enables a fine-grained tuning of performance and security.

6 ACCESS PATTERN PRIVACY

Memory access patterns might reveal information about the underlying plaintext. For instance, the number of input and output values of a data structure can potentially be revealed through memory access patterns. If the input is the name of a person and the output is her list of medical consultations then, typically, the length of the output should be hidden. Another example is a sorted array of encrypted values that is searched using binary search. Though it might be impossible to disclose what (precise) value is searched, the fact that the binary search algorithm stops at a specific element, e.g., the first element in the array, might prove valuable for an attacker. Hiding memory access patterns might cause significant performance penalties, eg. Corollary 1 states that it could be that a program needs to access just one memory location for a given input, but to hide patterns we must access many many more locations since other inputs require to access them. Therefore one might decide to hide access patterns only for a certain section of code or conditionally based upon values of a set of variables.

Let us discuss the case of searching a value in a data structure, which covers a variety of important applications such as database queries. For a search on an arbitrary data structure denote by PAT the set of all possible access patterns for a fixed set of memory locations and any input that one might search for. A single access pattern for a value $val(v)$ is a sequence $L(val(v)) := (l_0, l_1, ...l_{n-1})$ of memory locations that are needed to solve the problem by a specific algorithm, for example, for binary search l_0 on an array of length n could be the center element $n/2$, l_1 could be the $n/4$th element and so on. Note that access patterns might differ in length. They might not be unique for each value in general, however, for simplicity we assume for our scenario that they are. Access subdomain privacy concerns the scenario where an access pattern should be hidden if the value of the variable v to be searched belongs to a set C. Otherwise, the confidentiality of v must still be protected, but access patterns can be revealed.

Definition 3 (Access Subdomain Privacy). *A variable v with value $val(v) \in D$ is* access subdomain private *for a set of values $C \subseteq D$ with respect to a set of memory locations L, denoted by $ACC(v,C,L)$ if for any value $val(v) \in C$ an attacker cannot disclose the access pattern $L(val(v))$ with probability of more than $1/|PAT|$.*

The above definition implies that for a private value, we must access all memory locations that occur potentially in any of the access sequences. Moreover the length of the access patterns must not be disclosed by the number of memory locations accessed, e.g., by letting an algorithm access the same number of memory locations for all values in C. This can imply a significant increase in the running time. E.g. an access subdomain private algorithm can keep the number of comparisons carried out always the same, irrespective of the value to be found. A simple implementation of such an algorithm may perform a binary search for all values $v \notin C$, for which the access patterns do not need to be hidden. For the values in C, all elements are inspected, i.e., to find a value v in a sorted array A, v is compared with each value $a_i \in A$, returning an encrypted bit b_i being 1 if $v = a_i$ and 0 otherwise. Thus the access pattern is always the same for any $v \in C$. In this implementation searching for values in C takes linear time instead of logarithmic time.

Additionally, subdomain privacy can be extended to relax the requirement of not revealing anything about the access pattern. E.g., there are scenarios with confidentiality constraints where the approximate positions within an array that is being accessed can be revealed but not which index exactly corresponds to the item to be found. In this case, a subdomain private binary search only needs to hide the last steps. Such a scheme can exploit value range partitioning.

Compared to traditional access privacy where all values of a variable require the use of pattern hiding techniques, subdomain privacy imposes the corresponding performance penalty only on a subset of values.

7 EVALUATION

We have evaluated our method for a selected set of operations and the JOS scheme (Schneider, 2016). It should be clear from prior descriptions that the usefulness of all concepts allowing for a more fine grained description of privacy for variables depends heavily upon the privacy requirements of the data at hand. Thus, it is not possible to make precise general statements with respect to performance gains. However, our selected applications should give some guidelines stating well-founded examples from health-care and industry. What is evident from prior descriptions is that (statistical) subdomain and access subdomain privacy introduces some overhead, i.e., for a value we must determine whether it is to be kept confidential or not. This always requires at least one comparison using an encrypted value. Additionally, the compilation and execution of the program becomes more involved, since we must change code at runtime, i.e., a multiplication of two variables a, b might be performed using a protocol for encrypted values, a protocol for non-encrypted values or a protocol using one encrypted value and one non-encrypted value. The later source of overhead reflects only in local computation. In our evaluation we therefore compared against a system that does not support our presented privacy notions at all.

7.1 Subdomain Privacy

In Section 4 we described a scenario where common ("normal") values can be public, but exceptional values should be kept private and require special treatment. Algorithm 2 below shows a control algorithm with this behavior. It contains a simple branch for normal data ($v \leq 50$) and a more complex branch for abnormal data ($v \in [50, 1000]$).

Figure 1 shows the speedup for 10000 executions depending on the probability that a value v is normal for three scenarios: i) without subdomain privacy, ie. treating all values confidentially; ii) with domain privacy $DOM(v, [50, 1000])$ and iii) statistical subdomain privacy $STATDOM(v, [50, 1000], 95\%)$. Note, that even when all data is normal, we receive the data in encrypted form and must first check, if it is normal or not. This requires secure operations, ie. two comparisons of the value v with the lower and upper bound, i.e. 50 and 1000, yielding two encrypted bits. Since Algorithm 2 requires about seven comparisons (plus some computations) the speedup factor is inherently limited to roughly 10-20 – even if all data is normal. Using subdomain privacy yields a factor of about 800% speedup in this case. If all

Algorithm 2: Control Algorithm for Benchmarking.

```
1: **if** $v > 50$ **then**
2:     {Abnormal data, handle with $x, v$ confidential}
3:     **if** $v > 700$ **then**
4:         $x = a_9 \cdot \sin(v)$
5:     **else if** $v > 600$ **then**
6:         $x = a_7 \cdot \sin(v)$
7:     **else if** $v > 500$ **then**
8:         ...
9:     **else if** $v > 100$ **then**
10:        $x = a_2 \cdot \sin(v)$
11:    **else**
12:        $x = a_1 \cdot \sin(v)$
13:    **end if**
14: **else**
15:    {Normal operation, $v$ non-confidential}
16:    $x = a_0 \cdot \sin(v)$
17: **end if**
```

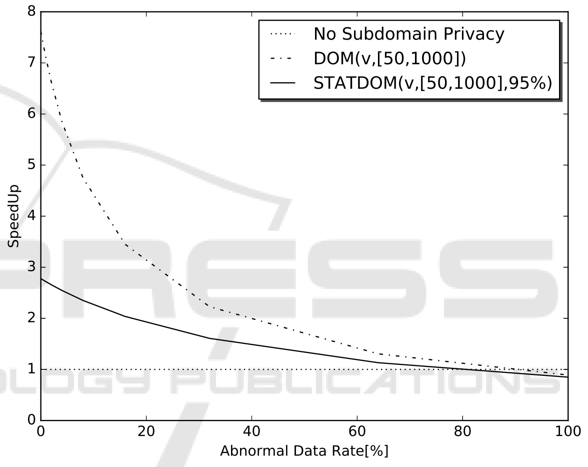

Figure 1: Speed up for Control Algorithm.

data is abnormal then subdomain privacy has a slight negative impact, ie. a slow down of about 10%, since all data is treated confidentially and the two comparisons to determine whether a value is normal or not comprise overhead. Statistical subdomain privacy behaves worse, since in addition to the two comparisons for determining whether a value is normal, it also requires a generation of a (secret) random number and comparison of the number with the percentage given in the definition of statistical subdomain privacy.

The second benchmark assumes 1 % of values are abnormal and focuses on the behavior of statistical subdomain privacy depending on its security parameter p_s. Figure 2 shows the speed up when using statistical subdomain privacy, i.e. $STATDOM(v, [50, 1000], p_S)$, for varying revelation probability p_S for disclosing whether $val(v)$ given that $val(v) \notin C$ in Algorithm 2. If we do not reveal any values, statistical subdomain privacy comes with

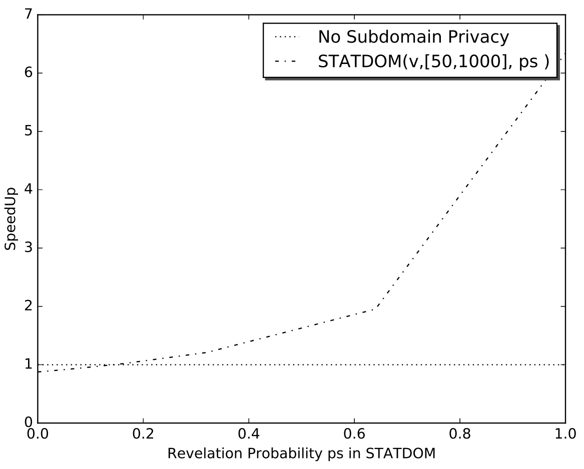

Figure 2: Speed-up for Control Algorithm for varying parameter for statistical subdomain privacy.

a slight run-time increase of about 10%. Good speedups are achieved for large values of p_S exceeding a factor of 6.

Next, we use synthetic clinical data of patients having some disease and healthy persons. We look for patterns, eg. we want to determine whether specific patterns like genes in the DNA correlate with a disease, or whether a specific pattern in body temperature over time is characteristic for a disease. We model the data by a binary sequence of 1000 bits and try to find 10 bit sequences of 10 bits each. For healthy patients we are allowed to decrypt the data and compute on plaintexts. For a specific pattern our algorithm simply checks from each position i whether the next 10 bits match the pattern. A minor concern is that in this case we have two variables, one being the health state bit $h(A) \in \{0,1\}$ of a person A and a second variable being the DNA $DNA(A)$ of person A represented by 1000 bits. So far, we have only discussed the notion of subdomain privacy for a variable v depending on the value $val(v)$ of variable v itself but not dependent on the value $val(w)$ of another variable w. The extension is rather simple: All we need is that a variable v should be kept confidential if a variable w is kept confidential. An alternative but generally less efficient approach is to concatenate variables, ie. $data(A) = h(A)|DNA(A)$ and to define subdomain privacy by only considering the value of the first bit, ie. the 1001st bit, yielding $DOM(data, [2^{1000}, 2^{1001} - 1])$. We went for the latter option, since it is supported out of the box by our implementation.

Figure 3 shows the speedup for 10000 persons depending on the probability that a person is healthy. Up to somewhat less than one per cent of sick people the speed up is more than two orders of magnitude. The maximum possible improvement is a factor of 100 for one per cent of sick people. Getting close to this ratio is only possible, since the duration of the secure comparisons to check if a variable can be revealed are insignificant compared to the costs of the computations that are done with the data afterwards. Given that there are only sick persons there is essentially no overhead.

In Figure 4 we assumed that 95% of persons are healthy and focused on the behavior of statistical subdomain privacy parameter. The maximum possible gain is a factor of 20 for a revelation probability of 100%, which we closely achieve. The decrease in speedup is exponential given a decrease in revelation probability.

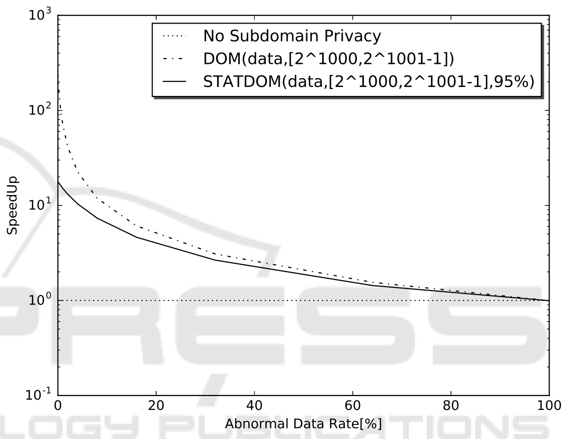

Figure 3: Speed-up for pattern matching.

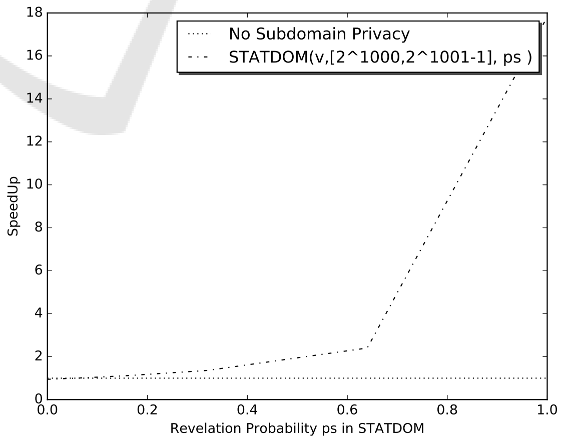

Figure 4: Speed-up for pattern matching for statistical privacy.

8 RELATED WORK

There is a rich literature on (automatically) translating code into equivalent code that can be executed in a SMC environment. Multiple existing SMC schemes have been combined into a framework that comes with its own programming language allowing the user to declare different kinds of so-called protection domains (Bogdanov et al., 2014). A protection domain serves as an abstraction of a set of SMC protocols. Another SMC compiler is provided in the Sharemind SMC framework (Laud and Randmets, 2015). The underlying motivation is maintainability which is achieved by introducing a domain specific language.

A variety of two-party protocols, i.e., garbled circuits, homomorphic encryption and hybrids, have been compared in (Ziegeldorf et al., 2015). The authors found significant differences in the running time. Building upon this finding, a mechanism has been proposed to automatically select the best protocol in order to maximize performance for two-party protocols using a cost model formulated as an integer program (Kerschbaum et al., 2014). Furthermore, a compiler has been introduced that allows the user to state whether a data item is public or private (Zhang et al., 2013). It also explicitly has a command for revealing a private value, i.e., making it public. In addition, it features explicit commands for executing commands in parallel, using threads of a thread pool. A formalized programming language for SMC based on Boolean circuits evaluated with the GMW protocol (Goldreich et al., 1987) has also been described and analyzed (Rastogi et al., 2014). Recently, several schemes have been combined, including Yao's garbled circuits, circuit randomization (Beaver, 1992), and boolean sharing (Goldreich et al., 1987), with conversion among them to adjust for different operations (Demmler et al., 2015). Both approaches also support mixed protocols, i.e., computations on encrypted and non-encrypted data.

There is a compiler for translating ANSI-C into secure two party computation based on garbled circuits (Holzer et al., 2012). It has a few (inherent) limitations, e.g., it can only handle bounded loops because the circuit grows with the number of iterations of a loop. Web-server operations can be made secure by translating (a subset of) C to secure code using partially homomorphic schemes as well as a small trusted computing base (Tople et al., 2013).

The database CryptDB enables the protection of data stored in a relational database while preserving the capability to run database queries (Popa et al., 2011). In this system, security also depends on the performed operations, i.e., certain operations such as comparisons require revealing more information about the confidential data. Searching on encrypted data has been treated by means of privacy-aware bucketization (Hacıgümüş et al., 2007). The idea is that in a client-server setting in which the server hosts a database, a client retrieves all values of a bucket and decrypts them and searches within the decrypted data. Thus, the server performs a preselection of values. The paper also discusses the level of information disclosure due to bucketization and algorithms to minimize information disclosure (depending on the data). Parts of this work also discuss search but in the context of memory access patterns. However, we do not send the entire content of a bucket to a client. Furthermore, our definition of value range partitioning explicitly reveals parts of a value. This is also not done in privacy-aware bucketization. Memory access patterns can also be hidden using ORAM, eg. (Stefanov et al., 2013; Bindschaedler et al., 2015). However, it comes at least with $\log n$ overhead, where n is the memory size (Goldreich and Ostrovsky, 1996).

Non-interference (Goguen and Meseguer, 1982) is the property of a system that its behavior and non-confidential output, observed by an unprivileged user (attacker), do not allow to infer confidential inputs. There has been a lot of work on determining whether programs achieve this property as well as constructing such programs, see (Sabelfeld and Myers, 2003) for a survey. In this context, our encrypted inputs and outputs can be seen as the confidential (high) variables. In contrast to our work, all possible input values are considered equally sensitive. Most closely related to our work is quantifying information flow (Clark et al., 2005; Clarkson et al., 2009) which quantifies, in terms of entropy and probability, how one can learn more about the value of confidential input variables by observing a program's behavior or its non-confidential outputs.

9 CONCLUSIONS

Perfect security – though highly desirable – remains often a distant dream in reality. Aside from errors in implementation, e.g. Heartbleed Bug, unproven assumptions, e.g. hardness of prime factorization, overhead related to security is often a key limitation. Whereas securing communication is a standard (and rather efficient) procedure using symmetric key cryptography such as AES supported in hardware, for computation on encrypted data it is impossible for many kinds of programs to achieve a 'reasonable' performance overhead compared to computations on plaintext. Therefore, the only way out is to trade off security and performance. Our paper addresses this need by introducing the concept of subdomain privacy, which formalizes the idea of decrypting a variable at run-time depending on the value it contains. Using subdomain privacy confidentiality of data can be defined on a more fine-grained level, which we believe is an important step to making computation on encrypted data feasible for a large range of applications.

REFERENCES

Beaver, D. (1992). Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology (CRYPTO)*, pages 420–432.

Bindschaedler, V., Naveed, M., Pan, X., Wang, X., and Huang, Y. (2015). Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security*, pages 837–849.

Bogdanov, D., Laud, P., and Randmets, J. (2014). Domain-Polymorphic Programming of Privacy-Preserving Applications. In *Proc. 9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–65.

Clark, D., Hunt, S., and Malacaria, P. (2005). Quantitative Information Flow, Relations and Polymorphic Types. *Journal of Logic and Computation*, 18(2):181–199.

Clarkson, M. R., Myers, A. C., and Schneider, F. B. (2009). Quantifying Information Flow with Beliefs. 17(5):655–701.

Demmler, D., Schneider, T., and Zohner, M. (2015). ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Proc. Network and Distributed System Security (NDSS)*.

Goguen, J. and Meseguer, J. (1982). Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*, pages 11–11.

Goldreich, O., Micali, S., and Wigderson, A. (1987). How to play any mental game. In *Proc. of 19th Symp. on Theory of computing*, pages 218–229.

Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473.

Hacıgümüş, H., Hore, B., Iyer, B., and Mehrotra, S. (2007). Search on Encrypted Data. In *Secure Data Management in Decentralized Systems*, pages 383–425.

Holzer, A., Franz, M., Katzenbeisser, S., and Veith, H. (2012). Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 772–783. ACM.

Kerschbaum, F., Schneider, T., and Schröpfer, A. (2014). Automatic Protocol Selection in Secure Two-Party Computations. In *Applied Cryptography and Network Security*, pages 566–584. Springer.

Laud, P. and Randmets, J. (2015). A domain-specific language for low-level secure multiparty computation protocols. In *Proc. of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1503.

Paillier, P. (1999). Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology–EUROCRYPT'99*, pages 223–238.

Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. (2011). CryptDB Protecting Confidentiality with Encrypted Query Processing. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100.

Rastogi, A., Hammer, M. A., and Hicks, M. (2014). Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 655–670. IEEE.

Sabelfeld, A. and Myers, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19.

Schneider, J. (2016). Lean and fast secure multi-party computation: Minimizing communication and local computation using a helper. *13th Int. Conf. on Security and Cryptography(SECRYPT)*.

Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. (2013). Path oram: An extremely simple oblivious ram protocol. In *Proc. of the SIGSAC conference on Computer & communications security*, pages 299–310.

Tople, S., Shinde, S., Chen, Z., and Saxena, P. (2013). AUTOCRYPT: Enabling Homomorphic Computation on Servers to Protect Sensitive Web Content. In *Proc. 20th SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1297–1310.

Zhang, Y., Steele, A., and Blanton, M. (2013). PICCO: A General-Purpose Compiler for Private Distributed Computation. In *Proc. 20th SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 813–826.

Ziegeldorf, J. H., Metzke, J., Henze, M., and Wehrle, K. (2015). Choose Wisely: A Comparison of Secure Two-Party Computation Frameworks. In *Proc. IEEE Symposium on Security and Privacy Workshops (SPW)*, pages 198–205.