1. Introduction

Every time we give away some information, we put at risk the safety of our or other people’s interests. It is important to keep secrets, but the statistical analysis of sensitive data can often be used for good intentions. There exist computer systems that are built to solve this problem.

A secure multi-party computation system called SHAREMIND [BLW08] is able to process secret data without disclosing. It uses the additive secret scheme in the ring \( \mathbb{Z}_{2^{32}} \) and is proven to give a strong privacy guarantee in the honest-but-curious security model. The system resembles a hybrid virtual machine – it contains a processor unit that can sequentially and in parallel execute a number of operations on the given secret or public data. In fact, this allows us to create algorithms to describe the way the data should be processed. SHAREMIND already has an assembly programming language [Jag08] to serve this purpose. However, complex algorithms are very difficult and tedious to program using low-level instructions.

To relieve the burden of crafting advanced algorithms we decided to design a higher level privacy-aware programming language called SecreC (pronounced as ‘secrecy’). The syntax of the language is based on C, but it omits several features and adds some new ones. Most importantly, we decided not to support pointers because of our concern about their unknown effect on the privacy of the overall system. Programs written in SecreC translate into existing SHAREMIND assembly and then run on the SHAREMIND virtual machine to accomplish data mining tasks. The design of the language in question is a compromise between various comfort and security factors. In this seminar paper we will review the main principles and features of SecreC, and discuss its good programming practices with security in mind.
2. Language principles

2.1. Data privacy and flow

Every piece of information can be a secret or common knowledge. Let us denote secrets as private data, and common knowledge as public data. It is clear that in order to guarantee the privacy of our secrets these kinds of data may not be mixed together in the public computation environment (Figure 1). Otherwise, one could witness the secrets while working with common knowledge. The lack of control mechanisms for limiting the flow of private data into the public data domain is a great security risk that has to be taken care of.

To reduce security risks there has to be a distinction between how public and private data is used. It is possible to construct an alternative hybrid execution model, where public and private data is processed in separate environments (Figure 2). The environments can be two computing systems that are specially built to handle the specific type of data. In a setup like this it is crucial to limit the publishing of information derived from private data into public data domain. While original secrets should never be published, some aggregated values such as a median or a sum can usually be published without leaking too much info. Therefore some strict means of controlling the data flow are mandatory.

---

**Figure 1.** The flow of public and private data in the standard execution environment.

**Figure 2.** The flow of public and private data in the hybrid execution environment.
The concept of the hybrid execution model is a generalization of the SHAREMIND virtual machine which in fact may be considered a hybrid virtual machine. While the public data and basic flow control is handled in the assembly interpreter (the public virtual machine), the private data is processed in the SHAREMIND processing unit (the private virtual machine) by core privacy-preserving computation protocols. The data can be processed using assembly instructions for all SHAREMIND protocols, the public data manipulation and flow control. While public data is stored and processed in public memory, the private data is computed on a special private stack in secret-shared form. Instructions for core protocols read the input from the stack, execute corresponding protocols and push the results back on the stack.

The described system contains all the necessary parts for the more secure model. However, it lacks the ease of algorithm implementation and the means of data flow markup. The design of SecreC aims at simplifying the programming task and preventing the developer of privacy-preserving data mining algorithms from making trivial privacy leaks. Furthermore, SecreC enables security analysis of private data flow by tracking it.

Additionally, all decisions during the program flow in the hybrid execution model have to be done on the public data. Otherwise, it would be necessary to publish the observable secret value, which is a security threat.

### 2.2. Vectorization

The SHAREMIND framework is known to be more efficient when executing several operations in parallel. This is due to the internal architecture of SHAREMIND – during each operation the system gets private inputs from the private stack, and then communicates over the network to perform some computations. Generally, the less system communicates over the network, the faster it is. Intuitively, to minimize the network traffic, we have to send a smaller number of bigger packets instead of a large number of smaller packets. A bigger packet takes almost the same amount of time to transfer than a smaller packet. This is due to the bandwidth overhead caused by the headers of each packet sent over the network.

On the basis of these properties it is advantageous to execute the same operation on a vector of data rather than on single values. The extensive use of operation vectorization is one of the main objectives in the SecreC design. The language must support intuitive and simple means of operation
parallelization and allow taking advantage of it by optimization of the algorithms.

3. Language features

3.1. The structure of a program
The SecreC language is a procedural imperative domain-specific language inspired by C. Generally speaking a SecreC program is a list of statements, that are executed one after another until the end or an error is reached. However, there are some syntax rules.

A program consists of a number of variable declarations and function definitions. In the very beginning of a program it is allowed to declare and initialize global variables. A variable declaration is one possible type of statement, and each statement in the program ends with a semicolon ‘;’.

Variable declarations are followed by function definitions. Every program must have at least one function called ‘main’. This is the first function to be executed after possible global variable declarations.

Program code may also contain line or block comments which are omitted by the SecreC compiler and do not affect the program flow in any way. Line comments begin with ‘//’ characters and consume everything up to the end of the same line. Block comments reside between the ‘/*’ characters and the first appearance of ‘*/’ characters allowing to include more than one line.

3.2. Data types and constants
The main innovation in the SecreC language is the use of secret data types. This is achieved by adding the visibility level to each fundamental data type. Data visibility can be either ‘public’ or ‘private’ and it is followed by the data type. Note that if no visibility specifier is given, then the public visibility is used by default.

Since the language is created with security in mind, the visibility level should not be confused with corresponding object oriented programming terms despite their similarity. Public visibility means, that data is stored in the public virtual machine and its value can be processed publicly. Private visibility means, that data is stored in the secret-shared form on the private stack or in a private register of the virtual machine. The shares of the value are distributed between multiple nodes (computers) of private virtual machine, so
that each node has only one share of the actual value. This is the basis of the underlying security scheme.

Each visibility level can be used in conjunction with certain fundamental data types. SecreC supports both public and private integer and boolean data types. For public visibility SecreC additionally supports the string data type. The void data type is also considered to be public for the sake of generality. In the following table we summarize the data types, their possible visibility levels and constants, if available.

Table 1. Data types, their visibility levels and constants.

<table>
<thead>
<tr>
<th>Visibility levels</th>
<th>Data type</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>void</td>
<td>none</td>
</tr>
<tr>
<td>public</td>
<td>string</td>
<td>“_”</td>
</tr>
<tr>
<td>public, private</td>
<td>bool</td>
<td>true / false</td>
</tr>
<tr>
<td>public, private</td>
<td>int</td>
<td>0 – 4294967295</td>
</tr>
</tbody>
</table>

3.3. Identifiers

All names in SecreC are identifiers. A valid identifier is a sequence of one or more upper or lower case ASCII letters, digits or underscores “_. An identifier may begin with a letter or an underscore but not the digit.

Another rule that the programmer has to consider when inventing his own identifiers is that they must not match any reserved keywords of the SecreC language. The reserved keywords are: void, string, int, bool, public, private, if, else, do, while, for, return, continue, break.

3.4. Variables

Variables are used to store data values, for example intermediate computation results. In order to use a variable, one must declare it first. This is done by writing the visibility specifier followed by the desired fundamental data type, which in its turn is followed by a valid variable identifier. For example the syntax for defining a public integer variable ‘x’ is:

```c
public int x;
int x;
```

Note that both declarations are identical because the public visibility specifier is optional despite the fact that it is recommended to be included for code
unification and clarity. It is also possible to declare and initialize a variable in one step using the following syntax:

```java
public int a = 3;
private int b = a + 4;
```

Currently this kind of variable initialization is only allowed if the right side of initialization is a constant, a variable or an expression, but not a function call.

There are two types of variables: global and local variables. Global variables are declared in the very beginning of a program outside all functions, and can be accessed anywhere in the code. Local variables are the ones declared anywhere within a function body or a block enclosed in braces `{}`. The scope of local variables is limited to the block where they are declared starting from the point of declaration. For example, if they are declared at the beginning of the function body, their scope is between the declaration point and the end of that function. This means, that if there exist two functions then the local variables declared in the first function cannot be accessed from the other function and vice versa. Additionally, if there are several blocks enclosed one into another then variables declared in the outermost block are also accessible in the inner blocks. This sets limits to variable declaration: if in some inner block a variable called ‘x’ is accessible (e.g. it was declared somewhere in the outer block), then it cannot be redeclared using the same name to avoid variable conflicts. The following program code example demonstrates discussed scope rules.

```java
private int a = 5;
void main () {
    public int n = 1; // ok
    while (n>0) {
        public int n; // wrong!
        public int b; // ok
        n = 0; // ok
        b = 1; // ok
    }
    b = 0; // wrong!
    a = 3; // ok
}
public int f (public int x) {
    public int n = 2; // ok
    public int a; // wrong!
    a = 4; // ok
}
```
3.5. Expressions

The SecreC language supports a number of operators that allow creation and evaluation of complex expressions. Most supported operators are binary. These include multiplicative, additive, comparison, logical and assignment operators. The composition of different constants, variables and operators forms an expression that can be evaluated.

After evaluation expressions have a resulting value, therefore as any other data unit, they are described with the visibility level and the data type. Since some expressions are public and the other ones are private then the question arises regarding what happens if expressions of different visibility level but same data type are combined into one single expression. Because the whole idea of privacy-preserving computation is keeping the secrets private, then publishing private data is out of the question. The solution is to convert expressions that contain private elements into private expressions. This is done by secret-sharing all the public elements and moving them to the private execution environment.

However, it is sometimes necessary to open the secrets to gain some knowledge for good intentions. As the private expressions are combined into a well built complex algorithm, it is possible to gain useful knowledge with high enough entropy not to compromise someone’s privacy. Additionally, to verify this it is necessary to have some kind of means of tracking the flow of private data in complex algorithms. SecreC introduces a special ‘declassify’ operator for explicitly publishing the private information and allowing the data flow analysis of algorithms.

Current state of function calls in expressions. Currently a function call cannot be part of an expression due some shortcoming in the compiler, that are yet to be fixed. Therefore, expression can only contain constants, variables or expressions with the same rules. Function calls can be used in conjunction with assignment operators as a right hand expression while assigning the result of a function to an existing variable e.g. `a = function_name(5);` As the development of SecreC goes on, the distinction will no longer be made after allowing function calls to be part of expressions.
Table 2. Operators and supported data types and visibility level combinations of arguments.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Left side data type</th>
<th>Right side data type</th>
<th>Left side visibility</th>
<th>Right side visibility</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign: =</td>
<td>all</td>
<td>all</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>add: +</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>sum: -</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>mult: *</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>matrix product: #</td>
<td>integer matrices</td>
<td>integer matrices</td>
<td>public</td>
<td>public</td>
<td>Not allowed</td>
</tr>
<tr>
<td>div: /</td>
<td>integers</td>
<td>integers</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>mod: %</td>
<td>integers</td>
<td>integers</td>
<td>public</td>
<td>public</td>
<td>Not allowed</td>
</tr>
<tr>
<td>or:</td>
<td></td>
<td></td>
<td>booleans, boolean matrices</td>
<td>booleans, boolean matrices</td>
<td>public</td>
</tr>
<tr>
<td>and: &amp;&amp;</td>
<td>booleans, boolean matrices</td>
<td>booleans, boolean matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>gt: &gt;</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>lt: &lt;</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>gte: &gt;=</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>lte: &lt;=</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>eq: ==</td>
<td>integers, integer matrices</td>
<td>integers, integer matrices</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
<tr>
<td>not: !</td>
<td>booleans</td>
<td>booleans</td>
<td>public</td>
<td>public</td>
<td>Allowed</td>
</tr>
</tbody>
</table>

3.6. Vectors and matrices
In addition to basic data types SecreC supports public and private matrices of integer or boolean types. Since matrices are nothing more but a generalization of vectors, the language does not distinguish the two. We state that a vector is
a one-column matrix and the language processes as such. Matrices play a crucial role in SecreC, because the virtual machine running the compiled code is known to be more effective when performing operations on arrays of data. The language offers simplified means for working with matrices.

To define a matrix of integers or booleans a pair of square brackets must be added right after the data type and include the expressions for dimensions between them like this:

```java
private int[4][5] a = 3; // current version
private int[][] a[4][5] = 3; // future version
```

These examples create a private matrix named ‘a’ with dimensions of 4 rows and 5 columns such that all its elements are initialized to value 3. The first example is the current way of doing it. As the language’s grammar was revised, the decision has been made in favor of the second example for the future version of SecreC.

The language also allows a fast and simple way for extracting or assigning matrix rows and columns. The following syntax is used for that:

```java
private int[4] x; // sugar for one column matrices
private int[5] y;
private int[4][5] z;
z[*][3] = x;  // assign a vector to 3\textsuperscript{rd} column
z[4][*] = y;  // assign a vector to 4\textsuperscript{th} row
x = z[*][2];  // extract the second column
y = z[1][*];  // extract the first row
```

A remarkably short way of doing element wise operations on vectors is the usage of standard arithmetical and logical operators on matrix expressions. This effectively removes the need for looping over the rows and columns using looping statements. Since the element wise operators execute the same operator on all matrix elements at the same time, it is also less time consuming and therefore much faster considering the nature of SHAREMIND. For example the code for squaring all the elements of the matrix ‘z’ is:

```java
z = z * z;
```

The private virtual machine would push the matrix twice on its stack, perform element wise private multiplication and pop the resulting shares back into the matrix variable ‘z’.
3.7. Functions

A function is a group of statements that is executed whenever it is called from some point of the program. A SecreC program may define any number of functions. The language supports function polymorphism allowing to define functions with the same name, as long as their parameter signatures are different. To define a function the following syntax must be used:

\[
\text{visibility type name (param1, param2, …) \{ statements \}}
\]

where

- visibility - the visibility of function return value (optional),
- type - the data type of the function return value,
- name - function name,
- param1, … - function parameters. Each parameter consists of parameter visibility level (optional), data type and a name. Parameters can be considered as predefined variables that will be accessible in the scope of function body.

Here is an example function:

```secrc
private int fn (public int a, private int b) {
    return a + b;
}
```

It takes two parameters, adds them, converting the public value to private value, and then returns the resulting private value. Next we want to use that function. We can call it from the main or any other function like this:

```secrc
void main () {
    private int x;
    x = fn(3,5);
}
```

Note that passing parameters to the function or returning a value is done by value, since SecreC does not support pointers. Most importantly, the data types as well as parameter and argument count must match. The visibility levels are allowed to differ at some extent. They work exactly as in the assignment operator, where the left side of an operator is a function parameter and the right side is the passed value to that parameter. One can pass public values to private parameters, but not vice versa.

A function may or may not have a return value. If not value is returned, then the return data type of the function is set to \texttt{void} and its visibility to public. If other return data type is specified, then the appropriate
value must be returned by adding a return statement at some point inside the function body.

The SecreC language also supports recursion. Functions can call themselves inside their own body. Extreme caution is advised in this case because of the risk of being trapped in a never ending recursion. This eventually causes the machine to run out of memory.

3.8. **Condition statements and loops**

A program is usually not limited to a linear sequence of instructions. During the execution process it may branch or repeat some steps. For that purpose SecreC provides flow control structures that serve to specify what, when and under which circumstances has to be done by the program.

**Conditional statements.** The *if* keyword is used to execute a statement or a block of statements if a condition is fulfilled. The corresponding syntax is:

\[
\text{if (condition) statement}
\]

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword *else*. It is used in conjunction with *if*:

\[
\text{if (condition) statement1 else statement2}
\]

The language also allows repeating some statements for a certain number of times or while a condition is satisfied. There are three possible ways to loop: using *while*, *do-while* and *for* constructions.

**The while loop.** Its format is:

\[
\text{while (condition) statement}
\]

and its purpose is simply to repeat the statement while the condition is true.

**The do-while loop.** Its format is:

\[
\text{do statement while (condition)};
\]

Its functionality is exactly the same as the while loop, except that *condition* in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of the *statement* even if the *condition* is never satisfied.
The for loop. Its format is:

\[ \text{for (initialization; condition; increase) statement} \]

and similarly to the while loop its main function is to repeat the statement while condition remains true. Additionally the for loop provides a specific location to contain an initialization statement and increase statement. This type of loop is specially designed to perform a repetitive action with a counter which is initialized and then increased after each iteration. The initialization part may only contain an expressions and not a variable declaration. All variables should have been declared in advance. The increasing of a value must currently be done in the old fashioned way:

\[ a = a + 1; \]

Skipping and breaking the cycles. To skip the current loop iteration and proceed with the next one SecreC provides the continue statement. The loop can be stopped completely by the break statement. For these to take effect they must reside within the loop’s statement.

4. Writing algorithms in SecreC

While knowing the main features of the SecreC language, writing efficient and secure programs still remains a non-trivial task. It requires a deep understanding of base principles of the underlying system. Based on the prerequisites discussed so far one can begin programming privacy-preserving data mining algorithms in SecreC. However, there are still some techniques that are important to keep in mind.

The first and most important technique to know is to declassify private data as little as possible. Data mining is closely bound with statistics where trends and aggregations are more important than individual values. This is good, because in privacy preserving multi-party computations no party is interested in losing control over their secret values. If there are many parties and many values, those can be used to find useful statistical facts. Therefore when programming the algorithms, one should gather a reasonable amount of data to be computed, keep the number of decisions depending on the private data as low as possible and use the data aggregation techniques that maximize the measure of uncertainty of the output result. A reasonable amount of data helps getting good statistical results while contributing to the uncertainty of
the final result. The more data is aggregated, the harder it is to derive individual values.

Since the first technique prohibits the decision making on private data it may seem, there is not much to do about it. However, there is a technique that allows decision making based on private data without disclosing. It is possible to replace if-then conditional statements with oblivious selection clauses. For instance consider the conditional statement:

```java
if (a) x = y; else x = z;
```

It can be represented as:

```java
x = a*y + (1-a)*z;
```

The third aspect to consider is the amount of vectorization used in the algorithm implementation. The less the program has to execute separate comparison and multiplication operations, the faster it will be in the long run. In reality the performance boost is significant even if there is very little to process.

We are going to observe a small example in SecreC. The following code counts needles in the haystack.

```java
public count (private int needle, private int[] haystack) {
    public int i = 0;
    public int count = 0;
    public int n; n = getRowCount(haystack);
    for (i = 0; i < n; i = i + 1) {
        private bool r = (haystack[i] == needle);
        public bool equal; equal = declassify(r);
        if (equal) count = count + 1;
    }
    return count;
}
```

This code is not secure nor efficient, because it leaks individual comparison results and it does every comparison separately. A better code would be:

```java
public count (private int needle, private int[] haystack) {
    public int n; n = getRowCount(haystack);
    // An indicator vector of ones and zeroes
    private bool[n] indicator;
    indicator = (haystack == needle);
    private int sum; sum = vecSum(indicator);
    public int count; count = declassify(sum);
    return count;
}
```
The last example is a much more secure and efficient version of the same code. It performs a vectorized comparison of the needle and the haystack and declassifies only the final result. The comparison results of individual values are kept private and never released.

5. Conclusion
In this seminar paper we have discussed the SHAREMIND framework capable of privacy-preserving computations. We have also described a privacy-aware programming language SecreC that is compiled and run on SHAREMIND. The hybrid execution model is one of the main principles of the language in question. The language is capable of handling public and private data and allows the private data flow analysis through an explicit ‘declassify’ operator. We have described most of SecreC’s main features and given the first reference point for the developers interested in implementing privacy-preserving data mining algorithms for practical purposes. Finally we have discussed some good programming techniques for better security and performance. The ongoing work on the SecreC language will improve its qualities and will hopefully make a difference in the real world.

6. References