Towards a Scripting Language for Automating Secure Multiparty Computation

Kung Chen¹, Tsan-sheng Hsu², Wen-Kai Huang¹, Churn-Jung Liau² and Da-Wei Wang²

¹ Department of Computer Science, National Chengchi University, Taiwan, email: {chenk,g9816}@cs.nccu.edu.tw
² Institute of Information Science Academia Sinica, Taiwan, email: {tshsu,liaucj,wdw}@iis.sinica.edu.tw

Abstract

In this paper, we report our ongoing work on a scripting language for automating the development of complex protocols for secure multiparty computation (SMC). Our scripting language models the participating parties in a peer-to-peer symmetric manner that each party holds their private data as well as any intermediate results jointly. Furthermore, we argue that an essential challenge of such a language for SMC is to enable users to express their requirements of "the right to use a piece of data" in a simple yet abstract way and its compiler can detect any potential breaches of these security requirements specified in a user script. To address this issue, we propose a three-level access attributes, namely public, private, and shared, that users can employ to express their security requirements by associating variables with these attributes. Our scripting language is implemented in a prototype compiler that generates Ruby code exploiting the distributed runtime of our SMC framework.

Categories and Subject Descriptors D.3.2 [Domain-Specific Languages]: Secure Multiparty Computation

General Terms Languages, Design, Security, Privacy

Keywords Secure Multiparty Computation, Scripting language, Security, Privacy

1. Introduction

Secure multiparty computation (SMC) involves computing functions with inputs from two or more parties in a distributed network while ensuring that no additional information, other than what can be inferred from each participant’s input and output, is revealed to parties not privy to that information. Since Yao’s pioneer work on secure two-party computation [14] and the extension to multi-party by Goldreich et al [4], there have been many proposals of protocols for specific problems as well as of general approaches for secure protocol development. Recently, language-based tools [5–7] are emerging to better support the systematic development of SMC protocols. We believe that such language-based approaches merit more attention from the research community of programming languages and compilers as their application to SMC have posed new problems to be explored.

Traditionally, the development of SMC has a strong tie with cryptography, and the ultimate goal is to pursue “ideal security”, namely nothing, not even the range of private data, is known to participating parties in the presence of adversarial behaviors. However, as argued by Du and Zhan in [2], “if the ideal security is too expensive to achieve, people might prefer low-cost solutions that can achieve security at an acceptable level.” Hence they proposed a different paradigm of SMC that aims to ensure that only a limited amount of information is revealed to a participant in the computation. For example, at the end of executing such an SMC protocol, each participating party holds the partial result such that only by summation of these partial results that we get the correct result. This scheme is commonly referred to as “additive sharing” of a secret among participating parties.

We support this line of investigation on SMC. Indeed, we have proposed an information theoretical framework to quantitatively measure the amount of privacy leakage in a protocol [10], and conducted empirical studies on such information theoretically secure protocols developed in a two-party setting [12, 13]. Besides, we have developed a distributed SMC framework for implementing secure protocols with the help of a commodity server in the Ruby language. Recently, we have also applied our approach to a real-world scenario of computing statistical functions over the patient data held jointly by two healthcare organizations [11].

Based on our previous works mentioned above, we strongly feel the need of a language-based tool for developing complex protocols of SMC so that many protocol irrelevant programming details can be abstracted away. Systems
such as Fairplay [6], SMCL [7] and TASTY [5] are all cryptographically based and cannot easily adapt to our needs. Therefore, we launched to develop a scripting language that enables users to write complex protocols for problems like all-pairs shortest paths concisely using common higher-level language constructs such as variables, arithmetic expressions, assignment, conditional expressions and for-loops.

The salient features of our language are a symmetric view of all participating parties and a three-level data access attributes, namely public, private, and shared, that users can employ to express their security requirements by associating variables with these attributes in a declarative manner. The scripting language is implemented in a prototype compiler that generates Ruby code exploiting the distributed runtime of our framework. Along with code generation, the compiler performs static analysis to detect any potential breaches of security requirements specified by variable access attributes.

2. The SMC Scripting Language

This section presents our SMC scripting language. First, we put our work in context by providing some background information and an overview of our SMC framework. Second, we illustrate the syntax and main features of our scripting language with an example, followed by a description of our approach to security specification and check in user scripts.

2.1 Background and Overview

We adopt the commodity-based approach to SMC proposed by Beaver [1] and assume that all participating parties are semi-honest [3]. Specifically, in the two-party setting, Alice and Bob will follow the protocol correctly, though they may store any intermediate results during the computation. Furthermore, they agree on a semi-trusted commodity server, Carol, who does not participate in their communication, and neither does it receive any data from both parties. All Carol does is delivery of random numbers in a specific way. During the execution of a protocol, data private to a party will never be revealed to the other party without being disguised by randomization. Intermediate results will be held by both parties jointly in the additive sharing manner.

The foundation of our scripting language is a rich library of protocols developed by a bottom-up, building blocks approach based on an information theoretically secure protocol for scalar product [9]. As shown in [8], such construction is theoretically secure as long as the initial building block is secure. We have implemented the library of protocols in a distributed SMC framework in the Ruby language. The protocol library, together with many utility functions, forms the runtime environment of our scripting language, which we refer to as the SMCS runtime.

Previously, to develop a new protocol, the user has to write the Ruby program for each party that utilizes the SMCS runtime and follows the building block convention to ensure security. Now, with the scripting language, the user can simply write a single piece of SMC script concisely using common higher-level language constructs. Given a user script, our prototype compiler generates executable Ruby code for each party that exploits the distributed runtime of our framework, as illustrated in Figure 1. For example, the user script, (foo.pdl), is compiled into two pieces of Ruby code, foo_1.rb and foo_2.rb, that will be executed on the machines of participating parties, respectively.

Although our scripting language is highly influenced by our SMC framework, we strive to make it as general and abstract as possible. Firstly, our language allows the declaration of multiple parties even though our current framework supports only two parties. Secondly, unlike previous work such as SMCL [7], we model the participating parties in a peer-to-peer symmetric manner that each party holds their private data as well as any intermediate results jointly.

Thirdly, irrelevant programming details such as data communication and the commodity server are abstracted away in our language. The user can simply focus on the data and computation required by the protocol under development. Finally, we propose a three-level access attributes, namely public, private, and shared, for modeling security requirements. By contrast, previously, data are usually classified into public and private levels only.

2.2 Syntax and Example

Except data security declarations, the constructs for variable declarations, expressions and statements in our language are pretty common. Hence, instead of formal specification, we shall use a nontrivial example script to illustrate its syntax and main features. The example is based on the real-world case study presented in [11], namely the measurement of disease burden of dengue fever (DF) and dengue hemorrhagic fever (DHF) in a specific time interval in Taiwan.

In this case, the measurement was conducted by combining the claims data collected by Bureau of National Health Insurance (NHII), Taiwan, with the notified Dengue case records kept by Centers for Disease Control (CDC), Taiwan. But, actually we can apply SMC to this case and obtain

![Figure 1. Compilation and Execution of SMC scripts](image-url)
In Taiwan, we have a numbering scheme for identifying citizens. 

In Taiwan, we have a numbering scheme for identifying citizens. The two boolean vectors, DF and DHF, are private data prepared by CDC for indicating the patients in the whole population. Finally, the total numbers of DF and DHF patients, total_DF and total_HDF, should be shared data since they are computed by the underlying SMC protocols with data from both parties.

The computation block (Line 11-26) begins by reading the private data from the two participating parties using four for-loops, respectively (Line 12-19). Recall that, after compilation of the script, NHI and CDC will have their own Ruby code to execute. Hence, such private data are input from the data owner and will not be revealed to others. The two shared variables, total_DF and total_HDF, are initialized (Line 20-21) using a conditional expression, respectively, and are computed using a for-loop in a similar way. Note that we cannot initialize the two shared variables directly with zero because constants are considered as public data that cannot be assigned to shared variables.

2.3 Security Specification and Check

An essential challenge of designing a language for SMC is to enable users to express their requirements of "the right to use a piece of data" in a simple yet abstract way and its compiler can detect any potential breaches of those security requirements. To address this issue, we propose a three-level access attributes, namely public, private, and shared, that users can employ to express their security requirements by associating variables with these attributes. Data that is global and known to each party is considered public while data that belongs to only one party and is not accessible to others should be declared private. On the other hand, shared data is a piece of data that can be used only when all parties agree. Thus, a consensus must be reached among the involved parties before using it. In SMC, any intermediate results jointly computed by the parties should be shared.

We argue that such a scheme of access attributes is easy to use and can be effectively enforced by a compiler. Specifically, given the access attributes for all variables declared in a user script, our compiler will perform a static analysis to check whether the right to use a piece of data is observed by all statements in the script. The check is based on a flat access lattice formed by the parties declared in a user script. In SMC, any intermediate results jointly computed by the parties should be shared.

Figure 2 displays the SMC script for computing the total numbers of DF and DHF patients who have claims data in NHI’s database. An SMC script in our language consists of three parts: party declaration, a sequence of data declarations and a block of computation statements. Here we have two parties, NHI and CDC (Line 1). The constant 23,000,000, denoted by N (Line 2), is about the number of citizens of Taiwan. Both NHI and CDC need to prepare vectors of such size as input data for our SMC script. The data used in the protocol are declared and grouped by their access attributes (Line 3 to 10). The two boolean vectors, HOS and OUT, declared under the NHI section are private data prepared by NHI and specify which persons have received outpatient service or have been hospitalized during the selected time period, respectively. Similarly, the two boolean vectors, DF and DHF, are private data prepared by CDC for indicating the patients in the whole population. Finally, the total numbers of DF and DHF patients, total_DF and total_HDF, should be shared data since they are computed by the underlying SMC protocols with data from both parties.

The computation block (Line 11-26) begins by reading the private data from the two participating parties using four for-loops, respectively (Line 12-19). Recall that, after compilation of the script, NHI and CDC will have their own Ruby code to execute. Hence, such private data are input from the data owner and will not be revealed to others. The two shared variables, total_DF and total_HDF, are initialized (Line 20-21) using a conditional expression, respectively, and are computed using a for-loop in a similar way. Note that we cannot initialize the two shared variables directly with zero because constants are considered as public data that cannot be assigned to shared variables.

2.3 Security Specification and Check

An essential challenge of designing a language for SMC is to enable users to express their requirements of "the right to use a piece of data" in a simple yet abstract way and its compiler can detect any potential breaches of those security requirements. To address this issue, we propose a three-level access attributes, namely public, private, and shared, that users can employ to express their security requirements by associating variables with these attributes. Data that is global and known to each party is considered public while data that belongs to only one party and is not accessible to others should be declared private. On the other hand, shared data is a piece of data that can be used only when all parties agree. Thus, a consensus must be reached among the involved parties before using it. In SMC, any intermediate results jointly computed by the parties should be shared.

We argue that such a scheme of access attributes is easy to use and can be effectively enforced by a compiler. Specifically, given the access attributes for all variables declared in a user script, our compiler will perform a static analysis to check whether the right to use a piece of data is observed by all statements in the script. The check is based on a flat access lattice formed by the parties declared in a user script. For example, the script in Figure 2 declares two parties, NHI and CDC; hence the compiler will derive the flat access lattice shown in Figure 3, where the ordering of the lattice indicates that Public is the top element and Shared is the bottom element while NHI and CDC are incomparable to each other.

Given the access attributes of variables and the access lattice, our compiler can derive the access attributes for all expressions occurred in a script in the typical manner. For example, the access attribute for the expression, \( \text{HOS}[0] \) and \( \text{DF}[0] \), is \( \text{Shared} \), which is obtained by taking the greatest lower bound of the access attributes of its two sub-expressions, namely NHI for \( \text{HOS}[0] \) and CDC for \( \text{DF}[0] \).

1 In Taiwan, we have a numbering scheme for identifying citizens.
The access attribute for conditional expressions is worth mentioning. Specifically, the access attribute of \((e_1 \ ? \ e_2)\) is the greatest lower bound of the access attributes of \(e_1\) and \(e_2\). In other words, the two lower bounds must be equal. For example, the conditional expression \((\text{True} ? 0 : \text{total} \ DF)\) will be rejected by our compiler.

Once we have the access attributes of expressions, we can use them to check the security of a statement accordingly. Due to space limitation, we only state the key rules we use. In an assignment statement, the access attribute of the right-hand side expression must be equal to that of the left-hand side target. In a for-loop, the access attribute of the loop index must be public. Similarly, any indices to a vector must be public. The read/write statements also deserve attention as they are side-effecting statements that may be exploited to reveal secrets. Obviously, the data to read should be either public or belong to the executing party. The rule for write statements is a bit more complex as they take a destination, a participating party or all parties designated by \(\text{All}\), as well as an expression as arguments, \(\text{Write(} \text{destination, expression}\text{)}\). Basically, we should not output the value of an expression to a "wrong" destination.

3. Discussion

Our scripting language provides a symmetric and abstract model of SMC. Although it is not a full-fledged one, with vectors, conditional expressions and for-loops, it can already enable us to develop many useful SMC protocols, such as set intersection, all-pairs shortest paths and transitive closures.

A key feature of our language is the three-level access attributes. Our compiler is highly driven by the access attributes information derived from a user script. Such information facilitates not only security checks but also code generation and optimization. In particular, if the access attribute of an expression is private, the code generator will generate only local computation on the specific party’s side, thus sparing the need to execute expensive protocols.

We have formally defined the rules that our compiler employs to perform security check and code generation. The next challenging task is to develop a theory to formally prove that these rules are sound in the sense that the generated code satisfies certain security properties. We believe that this will lead to some new research problems to explore.

Acknowledgments

This research is partially supported by the National Science Council, Taiwan, R.O.C. under grant number “NSC 100-2221-E-004-002-MY3”.

References