Secure Multiparty Computations into C++

Nikolaos Kofinas

1 SUMMARY

In this project, we created a Domain Specific Language inside C++ in order to handle the creation of functions for Secure Multiparty Computations. C++ it is one of the most powerful and commonly use programming language and our project proposed a way to include SMC inside C++. Now programmers can use it in order to include SMC inside their applications and create applications such as: a poker application with GUI which can be used from real users, an application that requires some dynamic input from the user, etc.

2 PROBLEM DESCRIPTION

Many applications need a way to execute functions which input are variables from different people. In some cases, these variables are non-sensitive and thus each party can send to each other their data but in some cases the data are sensitive. As an example we can give a game of poker where the cards of each party must be unknown to all other parties and thus this approach is not suitable.
In order to solve this problem, most application use a trusted authority which basically is a server that has no connection with any of the parties and each party can trust it. The server will collect all secret variables, calculate the function and then broadcast the result back to the parties. All application use this approach in order to calculate the shared functions but what happens if the trusted authority is not as trusted as the parties will want?
This question is relevant for application that handle sensitive information like a poker game where someone can play with real money. In such cases there is no guaranty that the trusted authority will not work together with a party and transmit to it all the sensitive information from all the other parties. Fortunately, there exist a way to compute a shared function without the use of a trusted authority.
The theory behind secure multiparty computations describes a way to execute a boolean circuit by transmitting it to all parties and then each party will execute only a part of the circuit without ever gather knowledge about the private value of another party. Boolean circuits are not the best way for someone to create a shared function and thus we need a tool which will help the user write a function in a more abstract way and then will translate this function into boolean circuits.

There are a various of different previous works, Wysteria, Fairplay, and Ansi-C, which have created such tools. Fairplay and Ansi-C are designed to work only between two parties and also can be used only to create standalone shared functions which then will be called from a host language. Wysteria is a more complete language that can handle both share and private computation and thus the user can write a complete application. The downside of Wysteria is that it support only basic functionality and cannot be used to create modern applications that require GUI, input/output, etc.
In this project we decided to embed Secure Multiparty Computations into an existing modern language which can be used to write modern applications. We chose to focus on C++ because it is one of the most common and mature programming languages.

3 Our Approach

In order to give to the user the capability to write a Secure Multiparty function which will directly compile to boolean circuits inside C++ we need to use the template meta-function capability of C++. This cannot be done by using native C++ code because we cannot intervene during compile time to the native code and thus, we cannot change the code that the user wrote into something that can be translated into boolean circuits. For this reason, we decided to create a Domain Specific Language which will have all the operators and the statements that the users need in order to create various SMC functions.
In order to create a DSL inside C++ someone can follow two approaches: 1) create a meta-parser which will take as input the DSL code as a string and during compile time will validate the code and generate the Abstract Syntax Tree or 2) Implement a DSL which will abuse the template meta-programming syntax of C++ and add all the crucial components inside various templates instantiations (more detailed definition and examples bellow).
The first approach seemed to be very suitable for our problem but as we found out the syntax of the meta-parser is very strange and in order to parse something simple (e.g. “a || b && c” where a,b, and c are free variables) you needed hundreds to thousand lines of code. Also the result of the parse is the Abstract Syntax Tree which will also require a lot of code in order to be handled by us during run time.
The second approach is easier to be implemented and also it is a lot more flexible compared to the first one. By using this approach, we can create struct templates which can hide all the underling functions that we need during run time. The downside is that the syntax of the DSL that we designed has no similarities compare to the syntax of C++ and if the user does not have any knowledge about templates will strangle to understand how to create a function.
In order for our DSL to work we designed a new type variable, the SMCvalue<type,id> which has as subclass the mySMCvalue<type,id> value. The type and the id are two templates variables that the user will assign in his code. The first one indicates the type of the variable (int or bool) and the second is the id of the value. The id is used to distinct all the parties/users which will take part into the SMC and each of them needs to be assigned of a different value as id. Also, the user must assign an IP address to each party, including himself. Additionally, it must assign its own secret value to the mySMCvalue value. The ip and the value assignments can be done dynamically during run time (e.g. obtain the IP addresses from a GUI interface).

The variables, types and statements of our DSL are defined bellow:

$$
\begin{aligned}
E ::= & Int, Bool, Var, \\
& Plus < E_1, E_2 >, Minus < E_1, E_2 >, Greater < E_1, E_2 >, Lesser < E_1, E_2 >, \\
& Equal < E_1, E_2 >, If < E_1, E_2, E_3 >, Return < E_1 >, Seq < E_1, E_2 >, \\
& For < \#loop, ForVar, E >, Id < Var >, Set < Var, E > \\
Var ::= & SMCvalue < type, id >, mySMCvalue < type, id >, \\
& sharedSMCvalue < type, id >, forSMCvalue < type, id >, idSMCvalue < id >
\end{aligned}
$$

In order to demonstrate the syntax of our DSL, lets assume this simple expression "$party_1.v + party2.v$" . By using our DSL this expression will have the following form: "$Plus < SMCvalue < int, 1 >, SMCvalue < int, 2 >>$". This syntax is different compared to C++ but we can implement the reduction relation for the DSL, by simple creating an evaluation meta-function. Before we continue to a more detailed description of our interpreter, we present a complete working function that calculates which of the two parties has the biggest value. We will use this example in order to illustrate the interpreter.

```
typedef mySMCvalue<int, 1> s1;
typedef SMCvalue<int, 2> s2;
s1 v1;
s2 v2;
v1.value = 2;
v1.ip = myip;
v2.ip = hisip;
typedef If<
    Greater<s1,s2>,
    Ret<Id<s1> >,
    Ret<Id<s2> >
> func;
int idOfWinner = wrapper<func>(v1,v2);
```

In this example the user must define two types because he creates a function for two parties. Then, he creates a function which as we can see is just a typedef of an expression. Finally, he call the wrapper function and he uses as template argument the function that he created and as inputs for the functions the actual variables of the parties.
During compile time, the wrapper first will create an initial environment which will contain s1 and s2 and then it will take the function and it will call:

```
eval<func,env>
```

Then the eval will match the function to an expression, statement or variable and then the evaluation will continue as an in-order traversal of a tree.
In order to evaluate each individual expression/statement/variable we used template matching which can match the current template input to the correct evaluation function. For example lets take the expression $Plus < SMCvalue < int, 1 >, SMCvalue < int, 2 >>$ and this eval meta-function:

```
template <typename Expr1, typename Expr2, typename Env>
```

```
struct Eval<Plus<Expr1,Expr2>, Env> {
   Eval<Expr1,Env> ...
   Eval<Expr2,Env> ...
...
}
```

This meta-function will be called when we need to evaluate the Plus statement and it will first call the eval for the left expr and then for the right expr. When the evaluation of the right sub-expression finishes it will evaluate the Plus and return the appropriate result.
Additionally, we may need to call for a statement a different eval meta-function and we can do that by abusing the template matching functionality. Let's take as an example sequence statement together with the set statement. When the left sub-expression of the left statement is a set statement we need to update the environment for the right sub-expression. For these reason we can create two different eval meta-functions specialized for each case:

```
template <typename Expr1, typename Expr2,  typename Env>
struct Eval<Seq<Expr1,Expr2>, Env> {
...
}
template <typename Expr1, typename Expr2, typename Expr2,  typename Env>
struct Eval<Seq<Set<Expr1,Expr2>,Expr3>, Env> {
Eval<Set<Expr1,Expr2>, Env> -> get New env
Eval<Expr3, NewEnv> ...
...
}
```

Inside each evaluation meta-function we added some appropriate static error checker which validate if the sub-expressions are reduced to a "correct" type. For example, if we have the Plus statement we need both sub-expressions to reduce to a int type. If they reduce to anything else, we can produce a static assertion error which will stop compilation and produce a error to the user.
Furthermore, some errors are syntax errors or it can be captured by the above technique and in order to catch those errors we must create special evaluation meta-functions which will produce always an error. As an example lets take the case that someone tries to call Set for a non-setable variable (e.g. SMCvalues):

```
template <typename T, int uId, typename Expr1, typename Expr2, typename
    Env>
struct Eval<Seq< Set<SMCvalue<T,uId>, Expr1> ,Expr2>, Env,> {
... Produce error
};
```

4 Results

The implementation for our approach is semi-completed. While the interpreter for the DSL is implemented, we didn't implement the translation of the result into the boolean circuits. All the necessary functions are in place and instead of boolean circuits they produce a simple string in order to validate the correctness of our approach.

We tried to check the correctness of our interpreter in a lot of strange cases but we need to give our code to some external users in order to create their own functions. By doing this we can find several bugs or flows in our design of the DSL.
Furthermore, the error handling approach that we used, while it can produce the correct errors, it is not elaborating for the user because it does not provide an easy way to extract the exact place that the error occurred. When the gcc compiler evaluate a failed static assertion, it prints the exact instantiation of the template in which the error occured. An experience user can read this instantiation but the reality is that sometimes the instantiation is huge. Also, the gcc will not stop in the first failed static assertion but it will continue the compilation and because of that sometimes it just produce errors which are directly connected to the initial one. For this reason, some times the user must ignore many errors and find the first one (mainly on the top) which is the actual error.
Additionally, we do not know if the way that we approached the problem was the best one. As we discussed in the third section, there is only one other way (in our knowledge) in which someone can create a DSL which can be validated in compile time. While the syntax of our DSL is strange, it seems to work and to produce the desired functionality. Furthermore, the error handling that we implemented can probably be improved.

5 Conclusion

In this project we designed and implemented a Domain Specific Language inside C++ which can be used in order to write functions for Secure Multiparty Computations. As a future work we must give the implementation to inexperience user in order to test it and find any serious bugs. Also, we must implement the code for the boolean translation and also create a handler for the external program which will run the boolean code.

References

[1] Ben-David, Assaf, Noam Nisan, and Benny Pinkas. “FairplayMP: a system for secure multi-party computation.” Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008.

[2] B. Kreuter, ahbi shelat, B. Mood, and K. Butler, “PCF: A portable circuit format for scalable two-party secure computation.” in USENIX, 2013

[3] Rastogi, Aseem, Matthew A. Hammer, and Michael Hicks. “Wysteria: A programming language for generic, mixed-mode multiparty computations.” (2014).

[4] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein, âĂIJSecure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces,âĂİ 2011, http://eprint.iacr.org/