A New Way to Think About Secure Computation: Language-based Secure Computation

Florian Kerschbaum

SAP Research, Karlsruhe, Germany

Abstract. Assume two parties, Alice and Bob, want to compute a joint function, but they want to keep their inputs private. This problem setting and its solutions are known as secure computation. General solutions to secure computation require the construction of a binary circuit for the function to be computed. This paper proposes the concept of language-based secure computation. Instead of constructing a binary circuit program code is directly translated into a secure computation protocol. This concept is compared to the approaches for language-based information-flow security and many connections between the two approaches are identified. The major challenge in this translation is the secure translation of the program's control-flow without leaking private information via a timing channel. The paper presents a method for translating an `if` statement with a secret branching condition that may not be known to any party. Furthermore, that protocol can be optimized using trusted computing, such that the overall performance of a program executed as a secure computation protocol can be greatly improved.

1 Introduction

Assume two companies each have a sales database and they are interested in identifying common patterns using data mining techniques, but are afraid to reveal their database, since it contains information that provides them with a competitive advantage. In another scenario assume, that a multitude of companies is gathering at a provider and they would like to benchmark their performance, but they are afraid to reveal their performance indicators, since it could reveal their processes' weaknesses and open points of attack to the competitor. Both of these scenarios are of high interest in the business world and both of them can be solved with the same technique. Secure computation allows two or more parties to compute a common function such that both parties receive the result, but keep their inputs private (except what can be inferred from the result). In the first scenario the inputs are the databases and the common function is the data mining technique, and in the second scenario the inputs are the performance indicators and the function is some statistical function, e.g. average, computed over them.

The basis for secure computation is to express the common function to be computed as a binary circuit. Then each gate of the binary circuit can be executed securely and privately with a secure gate protocol. One can proof by induction that a binary circuit, even consisting only of exclusive-or and logical-and gates, exists for any binary input-/output-behavior. This generality leads to the generality of the solution, since any function can now be computed with a binary circuit.

Kerschbaum F. (2007).
A New Way to Think About Secure Computation: Language-based Secure Computation.
In *Proceedings of the 5th International Workshop on Security in Information Systems*, pages 33-42
DOI: 10.5220/0002423300330042

This paper suggests a new method of constructing secure computation protocols: language-based secure computation. The basic idea is to compile a secure computation protocol directly from the programming language and in doing so exploit the techniques used in the manual construction of specialized protocols by automating them. This approach carries the potential for greater speed of the secure computation protocols, as well as increased flexibility in specifying (programming) them. E.g. one could allow loops with a public loop condition, use a variable number of inputs from the parties (as long as this number is known to both parties) and use specialized protocols improving the performance, e.g. for strings.

The contribution of this paper is besides the introduction of the concept of language-based secure computation, the analysis of the main challenges that need to be overcome in realizing the approach and the investigation of one major problem identified. This paper will present a protocol for securely and privately computing an `if` statement, where the result of the condition expression may not be known to either party, and then present an improvement in running time using trusted computing.

The structure of the paper is as follows: The next section will review related work in more detail giving the necessary references. Section 3 describes the concept of language-based secure computation. Section 4 elaborates on the problem of translating the control-flow of a program and section 5 introduces the problem of an `if` statement with secret condition, as well as its solutions, the regular one and the optimized one using trusted computing. The conclusions are presented in section 6.

2 Related Work

2.1 Secure Computation

Secure Computation was introduced in [23]. It introduced the problem of computing a joint, public function $f(a, b)$ between two parties Alice and Bob where Alice privately holds a and Bob privately holds b. The problem was solved for general functions for two parties, but exemplified with the famous Yao's millionaires' problem. In Yao's millionaires' problem two millionaires want to compare their wealth, but do not want to reveal the exact amount to the other party. It therefore computes the "greater-than" function on private inputs. This paper also introduced the general mechanism of circuit construction for the function $f(a, b)$.

Since then the functions to evaluate each gate in the circuit have been generalized and optimized. Clever generalizations to the multi-party setting with malicious attacker have been found in the information theoretic setting [4] and the cryptographic setting [10]. Many more clever result have been found in subsequent research on optimizing the protocols with different settings (cryptographic or information theoretic) and different attackers (semi-honest and malicious) which are not listed here.

The need for more efficient protocols has been identified a long time ago [9] and several such protocols have been developed and published in the literature. Problems considered range over a wide variety, but the data mining community has been particularly active. A landmark paper here was [14] introducing the ID3 algorithm. It uses circuit construction protocols only as sub-protocols and optimizes the overall performance and communication.

2.2 Secure Computation Compilation

The problem of translating a program specifying the function $f(a, b)$ into a secure computation protocol has been addressed in [15]. The paper introduces the FairPlay compiler and programming system that compiles a programming language into a register free, feed-forward only binary circuit that is guaranteed to be oblivious. Next to arithmetic expressions, it provides programming language constructs: `if` statements, `for` loops, functions, arrays and variable assignments. There are some restrictions on these: Any expression can be used in the `if` condition, but both branches are always evaluated. The limit of `for` loops can only be static, not even public data. Functions are completely inlined and therefore no recursion is allowed. Basic data types are boolean and integer (ignoring enumerated).

2.3 Language-Based Security

From the vast research on language-based security, including safe programming languages [7, 12], proof-carrying code [17] etc., this paper is most interested in research related to confidentiality policies, especially preventing information flows [20]. An information flow occurs when the contents of one variable is influenced by the contents of another variable [6]. Usually variables are arranged into multiple security levels (which form a lattice) and the objective is to prevent information flow from higher levels to lower levels. Information flows can be direct (e.g. by assignment), implicit (i.e. following from the control structure) or via covert channels (e.g. timing channels). In [21] a type system to prevent direct and implicit flows is introduced which, of course, can be statically enforced. I.e. in such a typed language no one can write a program that violates the security policy by direct or implicit flows. JFlow is an extension to the Java language that enforces exactly such a language [16]. It also provides features such as type variables, run-time type checking and type inference to make writing programs easier. Type systems have been extended to also prevent information flows from covert channels. Especially timing channels and network messages which are interesting to secure computation have been addressed in [19]. [1] introduced the idea of cross-copying the branches of an `if` statement with secret condition.

Secure program partitioning addressed the problem of trusting a host to do the computation [24]. Every party defines a set of other parties which it trusts to compute with its data. Then the program is divided, such that only hosts trusted with the data do the computation. This ensures that data may only be compromised by these hosts and that untrusted hosts do not get access to the data. Nevertheless, differently from secure computation, it requires a trusted (third) party to exist to do the computation, otherwise the program cannot be compiled in accordance to the policy. Secure computation actually intends to replace that third party and do computation between mutually distrusting parties.

3 Concept

3.1 Data Classification

Similarly to the classification in information flow, we assign each variable in the common program source a label. This label is the set of parties that know or may know the contents of the variable. E.g. in two-party protocols (i.e. Alice and Bob) the labels are: $\langle Alice, Bob\rangle$, $\langle Alice\rangle$, $\langle Bob\rangle$, $\langle\rangle$. This naturally extends to multi-party computation and forms a lattice (as required for many analysis methods). In the remainder of the paper we will refer to variables and data with three adjectives:

1. *public*: known to all parties, e.g. with label $\langle Alice, Bob\rangle$ in the two-party case.
2. *private*: known to one party, e.g. with label $\langle Alice\rangle$ or $\langle Bob\rangle$.
3. *secret*: known to no party, i.e. with label $\langle\rangle$.

3.2 Compilation Process

The compilation process takes a common program source and translates it into two (or more for multi-party settings) protocol programs. The protocol programs then execute a secure computation protocol. As described in previous sections such a tool exists [15], but due to its complicated translation process, it is inflexible and the result often lacks performance. The proposed approach to translation adopts ideas from compiler writing and programming language research. Its model of computation is based on common programming languages which leads to higher flexibility and performance.

Each party in the computation has a set of variables in which it stores intermediate computation results. These can be additional variables introduced by the compilation process or correspond to variables in the common program source. Compared to regular compilation we can ignore register allocation and spilling if we compile the protocol programs in another high-level source language.

Then the common program source is translated into building block protocols. These building block protocols correspond to the machine instructions of regular compilation. The translation can be done via an intermediate language, if that eases translation. There should be building block protocols for all operations and statements, such as $+$, $-$, etc., and for each possible assignment of data classifications. I.e. an operation $x_{\langle Alice\rangle} + y_{\langle Alice\rangle}$ is translated into another building block protocol than $x_{\langle Alice\rangle} + y_{\langle Bob\rangle}$ and $x_{\langle\rangle} + y_{\langle\rangle}$. This paper will present the building block protocol for `if` on a secret condition in Section 5.1.

4 Control-flow

This section will highlight a major challenge in translating programs into secure computation protocols. A program as written in a programming language has a control-flow. The instantiation of the control-flow, the flow of a particular program run, may depend on the input data of the program. The problem is that if the control-flow supports executing a particular basic block a variable number of times (depending on input data), then the number of executions of that basic block “leaks” information about the input.

In order to obtain that information, the attacker needs to observe the control-flow of the program. He can do that in two ways:

- Locally, by inspecting the program counter (debugging or emulating the program).
- Remotely (and locally), by timing the program.

A key insight is that the control-flow of the resulting secure computation protocol programs is a transform of the common program. This transform is public and reversible, i.e. if an attacker is able to inspect the program counter of the protocol program, he can infer a virtual program counter in the common program. One could imagine that the control-flow could be split between the two parties, such that only one party has this equivalence, but this approach only works for a limited set of programs where that one party may indeed know the control-flow. And for interactive protocol programs, the control-flow of both protocol programs has to correspond, since each message sent must be received by the appropriate, corresponding receptor. This enables both parties in an interactive protocol to track the control-flow of the common program.

Control-flow obfuscation [22] intends to make the transform hard to analyze. Excellent results can be obtained against a static adversary, but a dynamic adversary that is able to execute the program in a debugger or emulator is much more powerful and no theoretically founded security results exist.

5 `if` on Secret Data

An important control-flow problem is a simple `if` statement, but on secret data, i.e. neither Alice nor Bob may know the result of the branching condition, because they could infer information about the inputs not inferable by the result. It is anticipated that in a reasonably complex program, such `if` statements are the rule, and not the exception.

5.1 `if` Protocol

The ideas from above can be composed into a language-based protocol for `if`. We consider the problem of executing the `if` statement on secret data, i.e. neither Alice nor Bob may infer anything about the result. We assume that the problem of computing operators using language-based protocols has been solved, i.e. such protocols exist.

The first observation is that since the condition is secret, due to the control-flow dependency all assigned variables in the branches are secret as well. Therefore we will outline how secret variables are to be stored. We use any 2-out-of-2 secret sharing scheme, i.e. both parties need to cooperate to reveal the secret. Operator protocols are then defined on the shares and will likely need to result in shares again. The actual choice of secret sharing scheme may depend on the operators used and be optimized to increase performance, but some candidates are:

- Exclusive-Or
- Modular addition

E.g. Alice may have the value 6 and Bob may have the value 3. The combined 3-bit secret would be $1 = 6+3 \mod 8$. Let x_A be Alice's and x_B be Bob's share of a variable x. Then, note that the conditional probability $Pr[x = c|x_A]$ that x has a certain value $c \in \mathbb{D}_x$ given share x_A is equal to the a priori probability $Pr[x = c]$, i.e. no party gains any additional information from its share.

$$Pr[x = c|x_A] = Pr[x = c|x_B] = Pr[x = c] = \frac{1}{|\mathbb{D}_x|}$$

We then assume that the condition for the `if` statement can be evaluated, such that the result is shared between Alice and Bob. E.g. let c be a boolean condition (i.e. $0 = false$, $-1 = true$), then from the evaluation of c Alice obtains c_A and Bob obtains c_B, such that $c = c_A \oplus c_B$. The protocol for the `if` statement, then reduces to an Oblivious Transfer where one party may switch the inputs and the other retrieves according to his share. This is the same protocol used for the evaluation of a logical and gate in [8], but the difference is in the message being retrieved.

```
if (c)
    b1
else
    b2
```

Fig. 1. `if` statement.

The translation of the `if` statement in figure 1 proceeds as follows:

1. Gather all variables being assigned in the "then" branch $\mathtt{b}_1$. Let $\mathbb{V}_1$ be this set.
2. Gather all variables being assigned in the "else" branch $\mathtt{b}_2$. Let $\mathbb{V}_2$ be this set.
3. Compute the union $\mathbb{V} = \mathbb{V}_1 \cup \mathbb{V}_2$ of the two sets.
4. Append to the "then" branch $\mathtt{b}_1$ assignments of the form `v = v` for all $\mathtt{v} \in \mathbb{V} \setminus \mathbb{V}_1$.
5. Append to the "else" branch $\mathtt{b}_2$ assignments of the form `v = v` for all $\mathtt{v} \in \mathbb{V} \setminus \mathbb{V}_2$.
6. In the "then" branch rename any assigned variable $\mathtt{v} \in \mathbb{V}$ to $\mathtt{v}_1$. I.e. the assigment of the form `v = exp` becomes $\mathtt{v}_1$ `= exp` and every subsequent use of that variable `v` in the branch is renamed to $\mathtt{v}_1$ as well. Denote the set of renamed variables $\mathtt{v}_1$ by $\mathbb{R}_1$.
7. In the "else" branch rename any assigned variable $\mathtt{v} \in \mathbb{V}$ to $\mathtt{v}_2$. Similarly rename subsequent uses of that variable and denote the resulting set by $\mathbb{R}_2$.
8. To the current translated protocol programs append code for evaluating the branching condition, such that the result is shared. Let Alice obtain c_A and Bob obtain c_B, such that the branching condition $c = c_A \oplus c_B$. Ensure that c_A and c_B are fresh variables, i.e. not used anywhere else in the program.
9. Recursively apply the translation to the two branches: First $\mathtt{b}_1$, then $\mathtt{b}_2$. Append the result of the translation to the end of the current translations. The branches are non-interfering, since they assign to differently renamed variables, i.e. they can be safely executed sequentially. This also implies that both branches will be executed in the final translated protocol.

10. Append code to Alice's protocol program that generates two messages m_0 and m_1. Let m_1 be the concatenation of all assigned variables from $\mathtt{b}_1$: $m_1 = v_{11}, \ldots, v_{n1}$ $\forall v_{i1} \in \mathbb{R}_1$. Similarly let m_0 be the concatenation of all assigned variables from $\mathtt{b}_2$: $m_0 = v_{12}, \ldots, v_{n2}$ $\forall v_{i2} \in \mathbb{R}_2$. Such a concatenation is called a variable store, since it allows later decomposition.
11. Append code to Alice's protocol program that generates a random number r_A of length $|m_0| = |m_1|$ (the same length as the two messages). Compute two messages $m'_0 = m_0 \oplus r_A$ and $m'_1 = m_1 \oplus r_A$.
12. Append code to Alice's protocol program that generates two messages: $o_0 = m'_{0 \oplus c_A}$, $o_1 = m'_{1 \oplus c_A}$.
13. Append code in Alice's and Bob's programs that does an Oblivious Transfer of one out of (o_0, o_1). Bob will obtain the message $o_{c_B} = m'_{c_A \oplus c_B} = m_c \oplus r_A$.
14. Repeat steps 10 to 13, but with the roles of Alice and Bob interchanged. Let Alice obtain o'_{c_A} and Bob choose r_B. The variable store for Alice's protocol program is then $o'_{c_A} \oplus r_A$ and for Bob's protocol program it is $o_{c_B} \oplus r_B$.
15. The translation continues with the next statement.

5.2 Trusted Computing Solution

The problem with the previous solution is running time. For every `if` statement both branches are always executed. Let $O(b_1)$ and $O(b_2)$ be the running times of branches b_1 and b_2, respectively. Then the running time of the `if` statement is the sum $O(b_1) + O(b_2) + \delta$ (δ is the time to execute the common code).

The motivation for executing both branches in secure computation derives from the attacker's ability to inspect the program at run-time, i.e. debugging or emulating. This enables him to trace through the program and determine the branch taken and, since the compilation process is public, he can determine the branch in the source program and the result of the condition. This contradicts the secrecy requirement of the condition. Now, if we remove the ability to inspect the program from the attacker, can we do better? If the program is executed in a trusted computing processor, it can no longer be inspected by anyone. A trusted computing processor is capable of receiving encrypted code and executing it privately, such that no one can inspect it. For this purpose it publishes a public key, such that software providers can create encrypted versions of their programs and securely deliver them to the clients. Nevertheless the trusted computing processor does not remove all side-channels an attacker can observe. Most notably, the program's timing is still observable and may reveal information about the input. In the only proposed solution for secure computation using trusted computing [3] this has been recognized, but not solved. The authors assume that the computation is "oblivious" for which the current solution is to use binary circuits which also execute both branches. In this section we will outline an algorithm that can achieve better performance.

First, recall the observations that can be made about a program to infer private inputs:

- messages
- timing

Two trusted computing processors can communicate confidentially, if they know the public key of the other processor or share a common trusted certificate authority. In this case, they can use any session key establishment protocol to generate a private session key they can use to hide the content of the messages. The content is then not observable to an attacker. The length of the message can be padded to a common maximum length, such that it does not reveal any information either. The only information that an attacker can gain from a message is that the fact that is sent and when, i.e. its timing. We assume that all building block protocols employ these techniques (padding and encryption).

The approach to secure the timing of the computation can be similar to padding the length. In the simplest case, one computes the maximum time a computation can take, then measures the time it actually takes and idles the rest before returning the result. This can be a very difficult approach, since even elementary operations, such as multiplication, may not take a uniform time to execute [13], and cache timing can depend on the access pattern of private data [2]. Our algorithm assumes that each building block protocol used in compiling the source program is oblivious, i.e. it is secure and of constant time.

For each building block protocol $\mathcal{P}$ construct a corresponding dummy protocol $\overline{\mathcal{P}}$ that has the same observable behaviour (i.e. timing and messages), but no effect on the computation result. This can e.g. be achieved by the above variable renaming (as in the `if` protocol), but then not using the result values. Assign each building protocol an unique element $\mathcal{P}_A, \ldots, \mathcal{P}_Z$. Several building block protocols may have the same element as long as they have the same observable behaviour, e.g. a protocols for computing the product or integer division of two secret inputs. The key is that for any protocol in $\mathcal{P}_A$ neither Alice nor Bob can differentiate which it is and cannot differentiate it from $\overline{\mathcal{P}}$. We write $\mathcal{P} \equiv \overline{\mathcal{P}}$ to denote that $\mathcal{P}$ and $\overline{\mathcal{P}}$ are indistinguishable (which includes encryption and padding as mentioned above).

The translation procedure for the `if` statement on secret condition is then:

1. Translate each branch $b_i \in \{b_1, b_2\}$.
2. For each branch $b_i \in \{b_1, b_2\}$ of the secure `if` statement, compute a sequence $S_i = s_{i,0}, \ldots, s_{i,n_i}$ of the building block protocols $\mathcal{P} \in \{\mathcal{P}_A, \ldots, \mathcal{P}_Z\}$ used. Then compute the supersequence $S = s_0, \ldots, s_\delta$ of S_1 and S_2. This can be done in time $O(nm)$, the product of the length of the two sequences.
3. Fill each branch with dummy protocols $\overline{\mathcal{P}} \in \{\overline{\mathcal{P}_A}, \ldots, \overline{\mathcal{P}_Z}\}$, such that they match the supersequence. I.e. for each symbol s_j of the supersequence have a matching symbol t_j with $t_j \equiv s_j$. Note, that the result of the computation remains unaffected, but both branches have an observable behaviour identical to S.
4. Create code for a protocol $\mathcal{C}$ that is a regular `if` statement with the two branches b_i padded to S. Obviously the condition needs to be evaluated securely. Let $\mathcal{C}_A$ be the code Alice's side of the protocol and $\mathcal{C}_B$ for Bob's.
5. Encrypt $\mathcal{C}_A$ with the public key of Alice's trusted computing processor ($E_A(\mathcal{C}_A)$) and $\mathcal{C}_B$ with Bob's ($E_B(\mathcal{C}_B)$).
6. Insert code in Alice's protocol program to execute $E_A(\mathcal{C}_A)$ in Alice's trusted computing processor and similarly in Bob's protocol program for $E_B(\mathcal{C}_B)$.

The computation complexity of this protocol is linear in the length $|S|$ of the supersequence, which is bounded between $\max(|S_1|, |S_2|) \leq |S| \leq |S_1| + |S_2|$. We

can therefore expect some speed-up by this protocol and only in the worst case it will deteriorate to the performance of the protocol without trusted computing.

6 Conclusion

The concept of language-based secure computation was introduced. The major challenge of securely translating the control-flow was exemplified with the secure `if` statement protocol and the advantages of language-based secure computation have been shown by an optimization on that protocol that requires the `if` statement to be translated directly. Many of the outlined challenges, e.g. comprehensive proofs and other control-flow problems, such as `for` loops with secret bounds, remain to be solved and are subject of future research.

References

1. J. Agat. Transforming out timing leaks. *Proceedings of the ACM Symposium on Principles of programming languages*, 2000.
2. J. Agat, and D. Sands. On Confidentiality and Algorithms. *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.
3. Z. Benenson, F. Gärtner, and D. Kesdogan. Secure Multi-Party Computation with Security Modules. *Proceedings of SICHERHEIT*, 2005.
4. M. Ben-Or, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *Proceedings of the 20th ACM symposium on theory of computing*, 1988.
5. D. Brumley, and D. Boneh. Remote Timing Attacks Are Practical. *Proceedings of the USENIX security symposium*, 2003.
6. D. Denning. A lattice model of secure information flow. *Communications of the ACM 19(5)*, 1976.
7. C. Fournet, and A. Gordon. Stack Inspection: Theory and Variants. *Proceedings of the 29th ACM symposium on principles of programming languages*, 2002.
8. O. Goldreich. Secure Multi-party Computation. Available at *www.wisdom.weizmann.ac.il/˜oded/pp.html*, 2002.
9. S. Goldwasser. Multi party computations: past and present. *Proceedings of the 16th ACM symposium on principles of distributed computing*, 1997.
10. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. *Proceedings of the 19th ACM conference on theory of computing*, 1987.
11. O. Goldreich, and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 1996.
12. J. Gosling, B. Joy, and G. Steele. The Java Language Specification. *Addison-Wesley*, 1996.
13. P. Kocher. Timings attacks on implementations of Diffie–Hellman, RSA, DSS and other systems. *Proceedings of CRYPTO*, 1996.
14. Y. Lindell, and B. Pinkas. Privacy Preserving Data Mining. *Proceedings of CRYPTO*, 2000.
15. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-party Computation System. *Proceedings of the USENIX security symposium*, 2004.
16. A. Myers. JFlow: Practical Mostly-Static Information Flow Control. *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.
17. G. Necula, and P. Lee. Safe Kernel Extensions Without Run-Time Checking. *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 1996.

18. O. Rabin. How to exchange secrets by oblivious transfer. *Technical Memo TR–81, Aiken Computation Laboratory*, 1981.
19. A. Sabelfeld, and H. Mantel. Static confidentiality enforcement for distributed programs. *Proceedings of the Symposium on Static Analysis*, 2002.
20. A. Sabelfeld, and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on selected areas in communications 21(1)*, 2003.
21. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security 4(3)*, 1996.
22. C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of Software-based Survivability Mechanisms. *Proceedings of the international conference of dependable systems and networks*, 2001.
23. A. Yao. Protocols for Secure Computations. *Proceedings of the IEEE Symposium on foundations of computer science* 23, 1982.
24. S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems 20(3)*, 2002.