Senior Thesis 2006
Covert Multi-Party Computation

Yinmeng Zhang
Advisors: Manuel Blum and Luis von Ahn

Abstract

We introduce an extension of *covert two-party computation* (as introducted by von Ahn, Hopper, Langford in 2005), to multiple parties. *Covert computation* is a stronger notion of security than standard secure computation. Like standard secure multi-party computation, covert multi-party computation allow n parties with secret x_1 through x_n respectively, to compute a function $f(x_1, \ldots, x_n)$ without leaking any additional information about their inputs. In addition, covert multi-party computation guarantees that even the existence of a computation is hidden from all protocol participants unless the value of the function mandates otehrwise. This allows the construction of protocols that return $f(x_1, \ldots, x_n)$ only when it equals a certain value of interest (such as Yes, we want to collude with each other T) but for which <u>no party can determine whether anyone else even ran the protocol whenever $f(x_1, \ldots, x_n)$ is not a value of interest.</u> Since existing techniques for secure function evaluation always reveal that all parties participate in the computation, covert computation requires the introduction of new techniques based on provably secure steganography. We suggest two possible generalizations of covert two-party computation, one which is weak but achievable by a general protocol, and a stronger one for which we claim no general protocol can exist.

1 Introduction

The basic assumption of cryptography is that other people are untrustworthy, and as little as possible should be revealed to them. At the same time, it is often desirable to compute with other people. *Multi-party computation* refers to the computation of some public function f on n inputs by n parties, each holding one of the inputs.

A classic problem in multi-party computation is voting: here, each voter has a secret input, namely his or her vote. The voters wish to compute the majority function on their inputs without revealing any more information than absolutely necessary. Some information is necessarily revealed; for example, once the winner of the vote is known, anyone can trivially see that a random vote is more probably for the winner than for any other candidate. However, it is interesting to ask if the computation can occur in such a way that nothing about the inputs is revealed *except* what can be determined from the answer. In fact, this can be achieved for any functionality, and is what is usually referred to by *secure multi-party computation*.

We can also hide other types of information. Steganography (etymologically, secret-writing) is concerned with hiding the computation itself from outside observers. That is, to any outside observer (one who does not know some secret key), all conversation will be indistinguishable from normal. Thus, only the participants in a multi-party computation need ever know that the computation even took place. A provably secure scheme for public-key steganography was invented by von Ahn and Hopper in 2004.

1.1 Covert Computation

Now, when Alice sends a steganographically encoded message to Bob, Bob decodes it to read the message. Interestingly, when Alice sends a normal message to Bob, Bob can still run the decoding algorithm, except that he will get junk as output. By definition of steganographic secrecy, Bob cannot distinguish between a normal message and one that contains an encoding except by distinguishing between the decodings. This suggests an interesting variation on the steganographic scheme.

We can modify the scheme so that the decoding of a normal messages will indistinguishable from random. Now, if we could somehow encode the computation so that all the hidden messages look random, Alice and Bob could run a computation together and not even be sure if the other is computing along or just holding a normal conversation.

Why do we care? Consider the classic motivating example for secure two-party computation, dating. The inputs are single bits; the first party's input is 1 iff she likes the second party, and likewise the second party's input is 1 iff he likes the first party. The function to compute is the AND of the inputs. Presumably, if a party did *not* like the other party, it will be impossible for them to determine the preference of the other party, since in either case the output is unchanged. There is a small problem with this example, which is that it is incriminating to initiate the computation in the first case. As we have just observed, a party who does not like the other party already knows what the output will be, and has no reason to compute. Thus, to truly protect the delicate feelings of the parties involved, it would be necessary to compute in such a way that neither party could tell if the other party was computing except if the output to the function was 1. These rather stringent requirements are formalized in the notion of *covert two-party computation*, introduced by von Ahn, Langford and Hopper in 2005. They also gave a general protocol to compute any functionality covertly.

The purpose of this research is to extend covert computation to multiple parties.

2 Infrastructure

Since we are concerned with hiding information, it is important to clarify what is known. We assume that all parties know a common circuit for the function that they wish to evaluate, that they know what role they will play in the evaluation, and that they know when to start evaluating the circuit if the computation is going to occur. Finally, we assume adversarial parties know all such details of the protocols we construct.

To be able to hide communications, we must define ordinary channels to hide them in.

Messages are drawn from a set D of *documents*. For simplicity, we assume that time proceeds in discrete *timesteps*. Each party P_i maintains a history h_i, which represents a timestep-ordered list of all documents sent and received by P_i. We call the set of well-formed histories $\mathcal{H}$. We associate to each party P_i a family of probability distributions $\mathcal{B}^i = \{B^i_h\}_{h \in \mathcal{H}}$ on D.

At each timestep, each party receives the messages broadcast in the previous timestep, updates h_i accordingly, and then draws a document $d \leftarrow \mathcal{B}^i_{h_i}$ to broadcast. Note that this document could simply be the empty message.

Of course, our real concern is not the ordinary communication but the hidden messages. It is our goal to make these hidden messages appear random to the parties computing until the end of the computation. Rather whimsically, we can picture the world as a room full of sleeping people snoring strings which are indistinguishable from random.

3 Formal Definitions (Semi-Honest Model)

Formally, an n-party protocol is the vector of programs $\Pi = (P_0, \ldots, P_n)$. The computation of Π is as follows: at each round, each P_i is run on its input x_i, the security parameter 1^k, a state s_i, and the history of messages exchanged so far. A message (possibly the empty message) is then broadcast and added to the history of messages. The *transcript* of the computation consists of all messages exchanged, each labeled with their timestamp and author. At the end of the computation, each party P_i halts with some output, call it $\Pi_i(x)$. We say that Π *correctly realizes the functionality* f if for some i, $Pr[\Pi_i(\bar{x}) \geq 1 - \nu(k)]$.

For each party P_i we define a *view*, $V^{\Pi}_i(\bar{x})$ consisting of everything P_i sees: the input x_i, private random bits, and the transcript. We also define the view of the eavesdropper, $V^{\Pi}_{\phi}(\bar{x})$ consisting solely of the transcript.

Secure multi-party computation makes it possible to *privately compute f*. Intuitively, this means any subset of the parties I that follow protocol cannot, even by pooling their resources, learn anything about the inputs of parties not in I. Let x_I be the vector of all the inputs of parties in I.

We say that Π privately computes f if there exists a probabilistic polynomial-time algorithm S such that, for any subset of the parties I,

$$\{S(I, x_I, f(\bar{x}))\}_{\bar{x} \in (\{0,1\}*)^m} \approx \{V^{\Pi}_I(\bar{x}), f(\bar{x})\}_{\bar{x} \in (\{0,1\}*)^m}$$

Steganography guarantees that the eavesdropper cannot distinguish the computation from normal computation,

$$\{V^{\Pi}_{\phi}(\bar{x})\}_{\bar{x} \in (\{0,1\}*)^m} \approx \mathcal{B}$$

In the framework of covert two-party computation, this guarantee was referred to as *external covertness*, as it guarantees that external adversaries, i.e. those who do not participate in the computation, cannot determine if a computation occured.

Covert two party computation additionally guarantees *internal covertness*, where the other participant in the computation cannot tell if the computation is occuring during the computation. Let $\Pi : \mathcal{B}_i$ denote the two-party computation where the ith party communicates as normal instead of following the protocol. Let $V_i^{\Pi,n}$ the view of party P_i at the nth step. For any input $\bar{x}$, at any step n excluding the final communication,

$$V_i^{\Pi,n} \approx V_i^{\Pi:\mathcal{B}_{|i-1|},n}$$

In a two-party computation, once a non-random answer is revealed, it is obvious that the other party must have been computing along. We express this by saying that for any probabilistic polynomial-time algorithm D, there is a corresponding probabilistic polynomial-time algorithm D' such that if D gives party P_i an advantage in distinguishing the view when the other party is participating and the view when the other party is not participating, D' gives the negligibly less than the same advantage in distinguishing the output of the computation from random bits. In symbols, for any x_1 and distribution on P_0's inputs X_0,

$$\mathbf{A}dv^{D}_{V_1^{\Pi}(X_0,x_1),V_i^{\Pi:\mathcal{B}}(X_0,x_1)}(k) \leq \mathbf{A}dv^{D'}_{f(X_0,x_1),U_l}(k) + \nu(k)$$

and similarly for party P_0 with input x_0 on any distribution of P_1's inputs X_1.

4 Weak Extension

We can simply carry the definitions from two-party computation over to multi-party computation. We guarantee

1. *External Covertness.* To any outside observer ϕ, for any input $\bar{x}$,

$$\{V_\phi^{\Pi}(\bar{x})\}_{\bar{x}\in(\{0,1\}*)^m} \approx \mathcal{B}$$

2. *Internal Covertness.* To any coalition of parties I, for any input $\bar{x}$, at any step n excluding the final communication,

$$V_I^{\Pi,n} \approx V_i^{\Pi:\mathcal{B}_{I^c},n}$$

3. *Final Covertness.* To any coalition of parties I, for any PPT D there exists a PPT D' and a negligible function ν such that for any x_I and distribution on the other inputs X_{I^c},

$$\mathbf{A}dv^{D}_{V_I^{\Pi}(X_{I^c},x_I),V_I^{\Pi:\mathcal{B}}(X_{I^c},x_I)}(k) \leq \mathbf{A}dv^{D'}_{f(X_{I^c},x_I),U_l}(k) + \nu(k)$$

4.1 Weaknesses

The key difference between two-party computation and multi-party computation is that in the first, computation is a binary occurence. There are either 0,1, or 2 parties actually participating in the computation; if the number is less than 2, certainly no computation is occuring; if it is 2, then the computation is occuring. For multiple parties, this distinction is not so clear. In many cases it would be useful for the protocol to be tolerant to a minority of the parties not participating. For example, if 99% of the parties are participating in a vote, then we would like the vote to go through. Here, though, while we guarantee very complete security, the protocol will give random output except in the case all parties are computing.

4.2 Application

This can still be useful in all-or-nothing circumstances where it is desirable that 100% of the parties are computing. Suppose Company A has been attacked by a hacker. Company A has strong incentive not to reveal their weakness, but preferably they would like to catch the hacker. Suppose rumor has it that $m-1$ other companies were also attacked; if the companies pool resources, the hacker is caught. Company A can initiate a covert multi-party computation which returns a long string of ones when all the inputs are verifiable data logs from the hacker's attacks, and a random number otherwise. If even one company does not cooperate, no one, inside or outside the computation can distinguish if the computation even occured. If everyone cooperates, the hacker is caught, and presumably, since all the companies are in the same boat, having the other companies be aware of the computation is acceptable.

4.3 Implementation

Since we already have the powerful tool of two-party covert computation on hand, it is actually quite simple to achieve the multi-party case.

In the usual secure multi-party computation, the protocol is roughly as follows

1. Split each secret input into m random parts and share among the parties.
2. Two-party compute on the random shares.
3. Recombine.

The protocol works because, once the secrets are split, all intermediate results are random, and thus no information is revealed until the final recombination. This can be simply modified to read

1. Split each secret input into m random parts and share among the parties.
2. *Covert* Two-party compute on the random shares.
3. Recombine.

If all messages are steganographically encoded, we will have external covertness for mpc. Here, again, all intermediate results are random, and thus, by the final covertness guarantee of two-party computation, no party can ascertain if the other party is really computing with him. This guarantees that the computation is covert at least until the answer is revealed. If all parties are computing, then once the computation ends, if the answer is non-random, all parties will know all other parties were computing along. However, if even one party was simply communicating as normal, all results of "computations" with him will be indistinguishable from random and independent of the other results. Thus, the final result will be random.

5 Strong Extension

Multiple parties raises the tantalizing possibility that it could *never* be necessary to reveal exactly who participated in the computation. That is, could we strengthen final covertness to say that, once we see a non-random answer, we only know that some proportion of the parties participated (for example, 99 of 100), but negligibly more?

First notice, that it is sufficient to consider the case of three parties. If a protocol existed for more parties, then the parties could use that protocol, modeling the excess parties themselves and computing a functionality that ignores those excess parties' inputs.

5.1 Three People Puzzle

Returning to the whimsical view of the world, we are now concerned with a room full of sleeping people, some of whom are actually sleeping. The conundrum is that, to protect the identities of those who are only faking being asleep (i.e. really computing), it is necessary to do the complement, that is, make it equally hard to guess who is really sleeping (i.e. communicating as normal). We give an informal proof that a general protocol for multi-party computation cannot exist.

Imagine we have three people, a *Faker2*, a *Faker1*, and a *Snorer*. *Faker1* has an input and wants to send it to *Waker2* without admitting that he's awake (this is basically an anonymous broadcast to *Faker2*, except harder than usual because Sleepers don't cooperate with the protocol, and yet their identity must be protected). In this very pared down version of the original problem, assume *Faker1* knows who *Faker2* is, so our only concern in this problem is *Faker2*'s view.

5.2 Sketch Proof

The intuition is that if the protocol existed something special must happen in the last round of messages.

- Assume for contradiction there exists an algorithm that both hides guilt and produces an answer. Then there exists one that works in the minimum number of rounds (expected time?).
- Consider the last round of communication: *Faker2* sees either two random outputs, or one random and the another encoding something important.

- Consider the following alternative algorithm: run all but the last round of the original algoirthm, then make up random numbers for the last line. If this algorithm fails on more than a negligible amount of the time, we can break the original algorithm.
- Run the original algorithm; then, *Faker2* can substitute something random for first one party and then the other party's output. It shouldn't matter if we change the *Snorer*'s output, but apparently a non-negligible amount of the time we will be unable to retrieve the answer if we model *Faker1* with a random beacon. *Faker1* is caught; the original algorithm does not work.
- But, if we can always shrink a correct protocol, eventually we get to a single line protocol, and that simply doesn't work. Contradiction.

6 Open Questions

It still remains to be seen if the informal proof outlined above can be formalized. We would also like to ask if there exists an interesting stronger extension of multi-party computation different from the one proposed here. Especially interesting would be security against malicious adversaries, which the current protocol is not secure against.

References

[1] L. von Ahn and N. Hopper. Public-Key Steganography. In: Advances in Cryptology – Proceedings of Eurocrypt '04, pages 323–341, 2004.

[2] L. von Ahn, N. Hopper and J. Langford. Covert Two-Party Computation In: Thirty-Seventh Annual ACM Symposium on Theory of Computing '05, pages unknown, 2005.

[3] C. Cachin. An Information-Theoretic Model for Steganography. Information Hiding, 2nd International Workshop, pages 306-318, 1998.

[4] N. Dedic, G. Itkis, L. Reyzin and S. Russell. Upper and Lower Bounds on Black-Box Steganography. To appear in Theory of Cryptography Conference (TCC), 2005.

[5] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game. *Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218-229.

[6] N. Hopper, J. Langford and L. Von Ahn. Provably Secure Steganography. Advances in Cryptology – Proceedings of CRYPTO '02, pages 77-92, 2002.

[7] A. C. Yao. Protocols for Secure Computation. *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, 1982, pages 160–164.