Stable Matching with PCF Version 2, an Étude in Secure Computation

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Science)

by

Benjamin Terner

August 2015

Approval Sheet

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computer Science)

[illegible]

Benjamin Terner

This thesis has been read and approved by the Examining Committee:

abhi shelat
Advisor

Mohammad Mahmoody
Committee Chair

Dave Evans
Committee Member

Accepted for the School of Engineering and Applied Science:

[illegible]

Craig Benson , Dean, School of Engineering and Applied Science

August 2015

To Mom

Everything I am is a reflection of the woman who raised me,
and everything I do is an honor to her memory.

Abstract

The classic stable-matching algorithm of Gale and Shapley and subsequent variants by, e.g., Abdulkadiroglu et al. [APR05], have been used successfully in a number of real-world scenarios, including the assignment of US medical school graduates to residency programs and students in New York, Norway, and Singapore to high schools and universities. One shortcoming of the Gale-Shapley family of matching algorithms is susceptibility to strategic manipulation by the participants. The commonly used paradigm to mitigate this shortcoming, employing a trusted third party to compute matchings explicitly, is outdated, expensive, and in some scenarios, impossible. This makes stable matching a natural problem for secure, multiparty computation (SMPC). Secure multiparty computation allows two or more mutually distrustful parties to evaluate a joint function on their inputs without revealing more information about the inputs than each player can derive from his own input and output.

We use Portable Circuit Format (PCF), a compiler and interpreter for Yao's garbled circuit protocols, to produce the first feasible, privacy-preserving protocol for stable matching. In doing so, we improve the theoretical bounds for stable matching constructions, develop global optimizations for PCF circuits, and improve garbling techniques used by PCF. We also analyze variants of stable matching that are used in real-world scenarios. Experiments show that after aggressive circuit optimization, our protocols scale with good performance for matchings with hundreds of participants.

Acknowledgments

No milestone is an individual effort; a large number of people have supported me and this work, as it is really the culmination of five years at Mr. Jefferson's University and not just one in graduate school, but only a few can be properly acknowledged here. First and foremost, thank you to my advisor abhi shelat for providing the opportunity and the guidance to learn. Next, thank you to Professors Dave Evans and Mohammad Mahmoody, my teachers and committee members. Thank you to Professors Marty Humphrey, Alf Weaver, and Kevin Skadron, whose guidance, advice, and help over the years always pointed in the right direction. Thank you to Ben Kreuter and Nathan Brunelle, who have become my friends and mentors. Thank you to Debayan Gupta and Joan Feigenbaum, my collaborators on stable matching. Lastly, thank you to Zach, my twin and teammate in life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Secure Computation

Consider two billionaires eating dinner at a very expensive restaurant. When they finish their meal, the waiter delivers a very expensive bill and expects a large tip. The billionaires begin to quibble over who should pay, and they resolve that the richer one should pay the bill and the other cover the tip. Assume that neither billionaire wishes to reveal his true net worth to the other, for fear that the other may reveal his holdings to the IRS. Can the billionaires accurately compare their true net worths to discover who is richer, and more importantly, who should pay the bill?

Andrew Yao proposed a very similar problem in 1982 [YAO82] (back when a million dollars was a lot of money), opening up a vast and fruitful field of research known as secure computation. Consider a situation in which two mutually distrustful parties wish to compute a joint function on their inputs, but do not wish to involve or do not have the luxury of knowing a trusted third party. Secure computation allows distrustful parties to jointly compute arbitrary functions while preventing any player or onlooker from learning more about another player's input than what he could deduce from his own input and output.

1.2 Stable Matching

The stable marriage problem, originally introduced by David Gale and Lloyd Shapley [GS62] in 1962, has become a very deep and fruitful area of research since its inception, and in 2012 Shapley and Alvin Roth won the Nobel Prize in economics for its theory and application. Consider a group of N men and a group N

women, all of whom desire a heterosexual partner for marriage, but without an effective or fair marketplace through which to choose a suitable mate. The stable marriage problem (or, as we will refer to it more generally, the stable matching problem), asks whether it is possible to match the men and the women such that no man A and woman B exist such that A is matched to a woman b whom A desires less than B, and that B is matched to some man a whom she desires less than A. If no pair of A and B exists, then the set of marriages is called *stable* since no pair of mates has incentive to break their bonds of matrimony and vow anew.

Stable matching is not only applicable to marriages, it applies to the class of markets in which effective pricing mechanisms do not exist or are difficult to create. In these situations, constructing an efficient market often involves building an online clearing house where participants in the market submit preferences and a matching algorithm is computed. Variants of stable matching include the stable roommate problem [Irv85], which involves matching members of a single set rather than two bipartite sets, and stable matchings with additional requirements, such as medical students seeking residencies with their spouses.

As a motivating example, we wish to apply stable matching to sorority recruitment at universities. Sorority matching, just like many real-world applications of stable matching, is very important to the future of a recruit. The sorority which she joins will affect her social and professional networks for the rest of her college education and beyond, just as schools and residency programs affect the social and professional networks for the rest of a student's life. When the stakes are so high, the process requires effective, robust mechanisms.

1.3 Secure Stable Matching

Stable matchings are extremely well-suited to secure computation because of privacy concerns regarding the participants' input preferences and strategic incentives to violate the privacy assumption. We show that the trusted third party model is outdated as well as dangerous. In addition, there is often no trusted third party to compute a matching! Collaboration among women is dangerous to the men, and the stakes are high enough in real-world matching situations – with potential benefit to all of the women involved – to incentivize cheating or collusion with the third party at the expense of the men. We show that it is possible to enforce fair play by protecting input privacy.

The attempt to implement stable matching illustrates some of the weaknesses in secure computation technology, but shows how close the field is to providing implementations in the real world. In the effort to develop an application for securely computing stable matching, we also developed general optimizations to PCF, our platform for secure computation.

Chapter 2

Garbled Circuits

2.1 Yao's Garbled Circuits

The garbled circuit protocol is an interactive algorithm between two players, whom we will name *Gen* and *Eval*. Adapting the notation of Lindell and Pinkas [LP09], we say that *Gen* and *Eval* will jointly compute a function $f(x,y) = (f_x(x,y), f_y(x,y))$, where *Gen*'s input and output are given as $(x, f_x(x,y))$ and *Eval*'s input and output are given as $(y, f_y(x,y))$. We will assume for the sake of simplicity that $f_x(x,y) = f_y(x,y)$, *i.e. Gen* and *Eval* receive the same output from the protocol, and denote it simply as $f(x,y)$. For the sake of simplicity, also assume that $x, y, f(x,y) \in \{0,1\}^n$ for some integer n.

Let $(G(1^n), E_k(m) = c, D_k(c) = m)$ be a private key encryption scheme such that $G(1^n)$ generates an encryption key k, $E_k(m)$ is an encryption algorithm using key k on message m that produces cipher text c, and D is a decryption algorithm that uses k to decrypt c and retrieve m.

Let OT be an oblivious transfer protocol between two players $P1$ and $P2$, where $P1$'s input is (m_0, m_1) and P_2's input is $b \in \{0,1\}$. $P1$'s output from the protocol is $\perp$ (indicating no output), and $P2$'s output is (m_b).

Yao's Garbled Circuit scheme is an interactive protocol in which *Gen* will *generate* a garbled circuit and *Eval* will *evaluate* it to discover the outputs. We will assume that *Eval* honestly relays the output to *Gen* when she has completed the computation. This assumption can be removed for the final construction.

2.1.1 A Single Gate

We will show how to garble a single gate from the binary circuit which *Gen* and *Eval* compute, drawing heavily from the explanation given by Lindell and Pinkas [LP09]. The construction of a full circuit follows inductively from the guarantees provided by the encryption of a single gate. Begin by considering, for example, an AND gate:

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: A Standard AND Gate

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.2: A Standard XOR Gate

Garbling the gate will require obscuring the inputs to each gate and encrypting the outputs such that knowing the output of a gate betrays no information about its true value. We will replace the inputs and the outputs with keys that are mapped to their true values, and the mapping will be kept secret. Begin by generating six encryption keys by repeatedly invoking $G(1^n)$ and name them $k_x^0, k_x^1, k_y^0, k_y^1, k_z^0, k_z^1$. We replace the above standard gate with a new "garbled" gate and then define a new function for evaluating its output.

x	y	z
k_x^0	k_y^0	k_z^0
k_x^0	k_y^1	k_z^0
k_x^1	k_y^0	k_z^0
k_x^1	k_y^1	k_z^1

Table 2.3: A Garbled AND Gate

x	y	z
k_x^0	k_y^0	k_z^0
k_x^0	k_y^1	k_z^1
k_x^1	k_y^0	k_z^1
k_x^1	k_y^1	k_z^0

Table 2.4: A Garbled XOR Gate

Notice the convention that the subscript of k maps it to a specific input variable or *wire*, and the superscript describes the value of that input from the table above, or equivalently the *semantic* value of the wire. We now define a new function for evaluating a gate. Whereas for a standard gate, we define the function

$$g(x,y) : \{0,1\} \times \{0,1\} \to \{0,1\}$$

we now define the new function

$$garbledgate(k_x^\alpha, k_y^\beta) : k_z^\alpha \times k_y^\beta \to k_z^{g(\alpha,\beta)}$$

for $\alpha, \beta \in \{0,1\}$, and $g(\alpha,\beta)$ defined by the truth table of the standard gate it computes. A single arbitrary garbled row is defined by

$$c_{x,y} = E_{k_x^\alpha}(E_{k_y^\beta}(k_z^{g(\alpha,\beta)}))$$

Or, more verbosely, a full garbled gate table is completely defined by the values

$$c_{0,0} = E_{k_x^0}(E_{k_y^0}(k_z^{g(0,0)}))$$

$$c_{0,1} = E_{k_x^0}(E_{k_y^1}(k_z^{g(0,1)}))$$

$$c_{1,0} = E_{k_x^1}(E_{k_y^0}(k_z^{g(1,0)}))$$

$$c_{1,1} = E_{k_x^1}(E_{k_y^1}(k_z^{g(1,1)}))$$

Given c, the function *garbledgate* is evaluated by decrypting c using both of the input keys to discover the output key.

2.1.2 Circuits Composed Of Gates

To garble a complete circuit, *Gen* will compute all of the cipher texts described in the construction above and send them to *Eval* for decryption using the input keys. *Gen* composes gates to construct a circuit by using the output key (denoted as k_2 above) of one gate as the input key to the next gate. Gates with fan-out of more than 1 will have their output keys used as input to multiple gates.

There are two glaring issues with the above scheme. First, if *Eval* knows all of the keys k, then *Eval* can decrypt every possible output key for every gate in the circuit. Second, *Eval* can learn the semantic value hidden by each key simply by the order of the cipher texts it receives for each gate.

To handle the first issue, we maintain the invariant that *Eval* knows only two of the input keys, k_0^α and k_1^β for each gate that she attempts to decrypt. If *Eval* attempts to decrypt a c for which she does not have both keys, she will receive $\perp$. Second, *Gen* can obscure which key *Eval* decrypts by randomly permuting the cipher texts for a gate before sending them to *Eval*. This hides the values of α and β from *Eval*, but it forces *Eval* to attempt to decrypt every cipher text. The scheme for both of these issues will be revisited and improved in Section 2.3.

2.1.3 Input Privacy

In order to evaluate the circuit, *Eval* must learn the keys that correspond to both *Gen*'s inputs and its own inputs. *Gen* trivially sends the keys for its inputs to *Eval*, sending $k_{x_i}^\alpha$ for the value α of every one of its inputs x_i, revealing nothing about $x_i = \alpha$. To learn the keys $k_{y_i}^\beta$ for each of its own inputs y_i while preventing *Gen* from learning *Eval*'s inputs, *Eval* and *Gen* perform an OT where *Gen*'s inputs are $(k_{y_i}^0, k_{y_i}^1)$ and *Eval*'s input is β. All of the OTs for *Eval*'s inputs can be performed in parallel.

2.1.4 Correctness

A formal proof of correctness for this scheme is deferred to Lindell and Pinkas [LP09], but a couple of notions for correctness are important to give here. First, *Eval* should decrypt a cipher text for which she has both input keys with negligible probability of error. Second, *Eval*'s probability of decrypting a cipher text for which she does not have both necessary keys should be negligible. The probabilities of error will be dependent on the encryption scheme chosen, but the probability of error can always be reduced thanks to a result by Dwork, Naor, and Reingold [DNR04].

2.1.5 Security

Rather than give a formal proof of security, we give a couple of notions of security that are important to this scheme.

Eval's input privacy is intuitively reduced to the security of the OT, since *Gen*'s view of the protocol is only of the OT messages passed.

Gen's privacy is proven by constructing a simulator that can generate the view of *Eval*. In this case, the simulator will not be able to correctly garble a circuit to evaluate $f(x, y)$, so it generates a "fake" garbled circuit that always evaluates to $f(x, y)$ which must be indistinguishable from a real garbled circuit in an actual execution of the protocol.

2.2 Free XOR

It is not necessary that all gate garblings be generated independently. Koleshnikov and Schneider showed that garbled XOR gates may in fact be completely dependent on their input wires, eliminating the necessity to transmit any information between *Gen* and *Eval* for their computation [KS08]. In addition, the construction is incredibly cheap to compute, requiring only a single XOR operation per gate.

Begin by revisiting the construction of a garbled gate in Table 2.4; notice that rather than generating k_z^0 and k_z^1 using $G(1^n)$, it is possible to make each k_z completely dependent on the input keys. Assume that there exists some $R \in \{0,1\}^n$, where n is the length of each key k. Now, let all $k_z^0 = k_x^0 \oplus k_y^0$ and, for all $i \in \{x, y, z\}$, let $k_i^1 = k_i^0 \oplus R$. We can easily see that we have a new garbled XOR gate in Table 2.5.

And now, in order for *Eval* to evaluate an XOR gate, she needs only to XOR her two input keys. R must be chosen once by *Gen* and held globally to be used for all gates, and *Gen* must keep R hidden from *Eval*, lest *Eval* learn extra information about the circuit. A complete argument for security is deferred to Koleshnikov and Schneider.

x	y	z
k_x^0	k_y^0	$k_z^0 = k_0^0 \oplus k_y^0$
k_x^0	$k_y^0 \oplus R$	$k_z^1 = k_x^0 \oplus k_y^0 \oplus R$
$k_x^0 \oplus R$	k_y^0	$k_z^1 = k_x^0 \oplus k_y^0 \oplus R$
$k_x^0 \oplus R$	$k_y^0 \oplus R$	$k_z^0 = k_x^0 \oplus k_y^0$

Table 2.5: A Garbled Free XOR Gate

2.3 Point and Permute

We digress slightly from giving optimizations for gates in order to introduce new notation that will resolve some security concerns and reduce the decryption burden on *Eval*. We identified in section 2.1.2 that the truth table for a garbled circuit must be randomly permuted to prevent *Eval* from learning the semantics of the gate by the order of the cipher texts. We also noted in section 2.1.4 that *Eval* should have negligible probability of correctly decrypting a cipher text for which she does not have the keys, which is a concern because *Eval* needed to decrypt all possible cipher texts. We now give a construction that allows *Eval* to only decrypt one cipher text per gate.

Consider a wire w_i within the circuit to be composed of (k_i^α, p), where p is a random *permutation bit* and k is the key that the wire holds, as defined before. When *Gen* encrypts a wire, *Gen* appends p to the end of k. When *Eval* decrypts the wire to learn k, she also learns p, and with two values of p from two input wires she can deterministically select the value to decrypt from the garbled gate they enter. She does this by simply selecting the $2 * p_0 + p_1$ entry in the garbled table to decrypt. This scheme is compatible with Free XOR above by defining every p_z^1 for k_z^1 to be $p_z^0 \oplus 1$. When computing the new wires for a gate, we have that for wire i, $w_z^0 = (k_x^0 \oplus k_y^0, p_x^0 \oplus p_y^0)$ and $w_z^1 = (k_x^0 \oplus k_y^0 \oplus R, p_x^0 \oplus p_y^0 \oplus 1)$.

Intuitively, because the permutation bits are chosen independently of the keys, this permutation of garbled gate tables makes each table look randomly permuted, and in fact even gates of the same type are permuted independently of each other. This trick requires the garbling to store the semantics of the zero key for each wire, and every output wire must have the semantics of the permutation bit revealed in order to derive the true outputs.

2.4 Garbled Row Reduction

While the Free XOR trick allows us to eliminate all communication for XOR gates and evaluate them using a single XOR, it does not help us reduce the computation or communication necessary for other gates. Garbled Row Reduction (GRR3), suggested by Pinkas, et al. [PSSW09], permits us to reduce the amount of information communicated and stored for every non-XOR garbled gate by one entry, transmitting only 3 table entries (hence the name GRR3). The intuition for this optimization is simple. Rather than generating the key for both output values (k_z^0 and k_z^1) from $G(1^n)$, we can directly determine the value of one k_z from the two input keys that are used to generate it, by computing $k = E_{k_x}(E_{k_y}(0^n))$. We define this k to be the first row sent by *Gen* for the gate, so *Gen* no longer needs to send that row. The other k_z is defined as determined by the Free XOR method.

We note here that in most current constructions, the double encryption over a key is computed as

$$E^s_{k_x,k_y}(m) = H(k_x||s) \oplus H(k_y||s) \oplus m$$

where $H(k||s)$ is some hash function keyed by k on input s, and s is some string completely unique to each invocation of the encryption. In some constructions, s is given as $Gid||c_0||c_1$, where Gid is the gate's number, also commonly referred to as a label, and the c values are the "external" values of the wire, given by the permutation bits concatenated onto the end of them. Here, rather than compute the encryption over some message k_z (where k_z takes the place of m above), k_z is simply defined to be the mask computed by XORing hashes of the input keys. In addition, because this key is defined to be the first row in the table, both c values are 0. (The value c for this output wire is also implicitly defined by this operation.)

Intuitively, this row reduction should maintain privacy because it reveals no new information to *Eval* about the keys. If the key in the first position of the table were defined to be the zero string, then *Eval* could always recover that row. However, *Eval* will only be able to compute the first row in this scheme if *Eval* has both input keys to that row of the gate, since she must use them to encrypt the zero string. A complete proof of security is deferred to Pinkas, et al. [PSSW09].

2.5 Half-Gates

Pinkas, et al.[PSSW09] also showed that it is also possible to send only two ciphertexts per gate (with minimal assumptions), but doing so came at the expense of not benefitting from free XOR. Zahur, Rosulek, and Evans [ZRE14] introduced half-gate garbling, which permits communicating two ciphertexts per gate, is compatible with Free-XOR, and works for any gate with an odd numbers of 0 outputs.

The mechanism is by garbling two half-gates whose XOR becomes the full gate's output key. Intuitively, we can use the technique from GRR to communicate one ciphertext of each half gate for free, then combine the two half-gates to communicate the possible four ciphertexts at the expense of two. Zahur, Rosulek, and Evans also proved this is a minimal scheme for known garbling mechanisms.

Two Half-Gates

The “generator” half-gate’s two ciphertexts are $H(B) \oplus C$ and $H(B \oplus R) \oplus C \oplus \alpha R$, where B is an input key, C is the output key, R is the circuit’s global xor-offset, and α is a boolean variable that is determined by the kind of gate being garbled. The “evaluator” half-gate’s two cipher texts are $H(A) \oplus C$ and $H(A \oplus R) \oplus C \oplus B$, where A is the other input key. Since both A and B are input keys, the first ciphertexts for both half-gates can be communicated for free by setting $H(X) \oplus C = 0^k$ (where k is the security parameter). When *Eval* learns both half-gate ciphertexts, she can XOR the two to determine her final gate. We note here that H will need the gate index as its second input, and because there are two half-gates, it must be incremented twice, once for each half-gate.

The challenge becomes for *Gen* to communicate enough information for *Eval* to learn her two ciphertexts. We show that *Gen* only needs to send *Eval* two messages. The first is $T_G = H(A) \oplus H(A \oplus R) \oplus (p_b \oplus \alpha_b)R$, where p_b is the permutation bit of the second key and α_b is a bit that depends on the gate’s truth table. The second is $T_E = H(B) \oplus H(B \oplus R) \oplus (A \oplus (\alpha_a R))$, where α_a is another constant dependent on the truth table. With this information and the knowledge of two keys *Eval* already has, she can derive the output gate. She sets $W_G = H(A) \oplus s_a T_G$, where s_a is the permutation bit on her version of key A, and $W_E = H(B) \oplus s_b(T_E \oplus A)$, where s_b is the permutation bit on her version of key B. The final output key is then $W = W_G \oplus W_E$. For *Gen* (who knows the values of R), $W_G = H(A) \oplus p_b T_G$, $W_E = H(B) \oplus p_b(T_E \oplus A)$, and the output key is $W = W_G \oplus W_E$.

2.6 Fixed-Key Garbling

Until now, the discussion of garbling has treated the hash function as an abstraction. BHKR [BHKR13] showed that if AES can be treated as a pseudo-random permutation, then fast garbling can be achieved by using fixed-key AES on the input keys. In this construction, the fixed key is public to both evaluating parties, and the key derivation function on both input keys is $\pi(K) \oplus K$, where $K = 2A \oplus 4B \oplus T$; in this equation, π is fixed-key AES, T is a *tweak* and consists of the gate index, $2A$ means that key A has been doubled, and $4B$ indicates B has been doubled twice; we defer a discussion of doubling to BHKR [BHKR13] except to note that it can be quickly accomplished with single left shift.

Half-gates are only encrypted with a single key, but this does not preclude us from garbling with fixed-key AES. For half-gates, the key derivation function simply becomes $\pi(K) \oplus K$ where $K = 2A \oplus T$, as given in its original construction [ZRE14].

Chapter 3

Portable Circuit Format

Starting with Fairplay [MNP+04], a number of systems have been introduced to perform secure computation of arbitrary programs [WLN+15] [Zah14]. The process of specifying a protocol for securely computing an arbitrary function generally involves three phases: specification in some high level language, conversion to binary circuit representation, and on-line evaluation by garbling the circuit. KsMB [KasMB13] provided the specification of a system that compiled arbitrary C programs to a compressed circuit format capable of expression computations requiring billions of gates, called PCF. Key to the format was the intuition that loops with fixed bounds in the circuit need not be unrolled until the last minute: at evaluation time. After HEKM [HEKM11] showed that large secure computations could be performed without enormous memory cost, PCF provided an extensible framework for compiling and evaluating very large circuits. However, it was still capable only of rudimentary analysis and could not perform full optimization over large circuits. In this chapter, we introduce improvements to the PCF specification as well as techniques to perform static analysis of circuits which achieve global data flow optimization on circuits consisting of trillions of gates, providing the means to analyze circuits larger than have been evaluated.

3.1 PCF Version 2

After KsMB [KasMB13], PCF was rewritten to improve its memory model and make the circuit format more extensible. Because no information about the improvements has been published, we now provide a short description of PCF's structure and the new full list of opcodes for PCF Version 2.

3.1.1 PCF V2 Structure

As a first step to generating circuits, the PCF compiler passes a C program – restricted such that no array access or loop bound may depend on the input – through LCC [Fra91] to produce the program's bytecode. The PCF translator then converts the bytecode to a PCF circuit before the optimizer removes unnecessary gates. The resulting PFC circuit can be evaluated by an interpreter which implements the PCF specification.

PCF's computational model follows a standard stack machine, where wires are allocated as memory addresses and new wires are allocated dynamically on an expanding stack with function calls. We include the simple caveat that the zeroth wire in every stack frame holds a pointer to the absolute zeroth wire, which holds the global condition wire. In addition, because the expanding and contracting of the stack may lead the PCF interpreter to attempt to re-use leftover values in the wire table as new functions are called on top of old ones, the LCC-to-PCF translator ensures that new wires are cleared (set to 0) before each use. The PCF optimizer removes unnecessary assignments through faint variable analysis.

PCF implements conditional statements and conditional function calls by computing a MUX over the output of every branch in the program, keyed by the conditional wires; just as in the last version of PCF, the global condition wire and a list of branch targets are updated every time a conditional statement is reached. The new PCF optimizer does not currently support inter-procedural data flow analysis, but it is capable of analyzing very large sequential programs if their function calls are inlined.

PCF may be extended by designing and implementing extra opcodes, but these are not necessary. It provides an interface via the call function for hooking special features or additional protocols; for example, inputs and outputs are designated and extracted by calling special functions provided by the PCF toolchain.

3.1.2 PCF V2 Opcodes

Table 3.1 describes the opcodes that comprise the PCF Version 2 specification. Descriptions of each operation are given in the table and examples are given in Figure 3.1. An example of the indirection operations is given in Figure 3.3.

3.1.3 PCF V2 Interpreter

The PCF interpreter was rewritten since KsMB [KasMB13] to go along with the improved compiler, optimizer, and new opcode specification, but the significant improvements are in garbling. PCF still uses lazy gate generation and pipelined evaluation [HEKM11], and now evaluates gates using fixed-key AES for encryption [BHKR13] and half-gates [ZRE14] for odd-parity gates.

bits	Splits an integer into its two's complement representation and stores it.
join	The (almost) inverse of **bits**, combines an unsigned integer into a single constant value.
gate	Emits a gate with truth table, input wires, and output wire.
const	Creates a constant value and stores it in the provided destination wire.
add, sub, mul	Arithmetic instructions that perform arithmetic on input-independent values.
initbase	Initializes the program's base pointer. Appears once, at the very beginning of the program.
clear	Set memory from baseptr to (baseptr + localsize) to 0, used to clear space for local variables.
copy	Copies a value from one position to another, x=y.
mkptr	Creates a pointer to a specified wire.
copy-indir	Dereference a pointer and copy the dereferenced value to a location (x=*y).
indir-copy	Copy a value to the location pointed to by a pointer (*x=y).
call	Calls a function given by fname and sets the new base.
ret	Returns from a function call, setting the return value (if one exists).
label	Assigns a label to an instruction address, used to name functions and bits of procedures.
branch	Executes a conditional jump on supplied wire.

Table 3.1: PCF2 opcodes and descriptions

3.2 Data Flow Analysis

PCF is a small, portable, and extensible tool for defining circuits. However, naïvely compiling PCF programs introduces a substantial amount of extra garbled gates, especially if the objective function does not need all of the bits in the standard available data types. We would like a way to indicate to the PCF compiler or optimizer which gates are important to our final output in order to remove unnecessary, expensive garbling from our protocols. For small computations or computations that do not conform to the standard data types provided by C, removing extra wires that result from compiling to a poorly fitting standard can substantially reduce circuit size.

Without writing a new programming language feature, this can be accomplished by forcing wires in the PCF circuit to become dead and then removing dead wires using data flow analysis. Wires can easily be forced dead by performing a simple bitwise AND with constants that contain a 0 in every unnecessary bit position, and constant propagation (explained in Section 3.2.3) can identify the 0 outputs and propagate them through the circuit. We remove forced-dead gates with faint variable analysis, including the masks

```
(BITS :DEST (1 2 3 ... 31 32) :OP1 100 ) // decomposes 100 into 32 wires, beginning at wire 1

(JOIN :DEST 1 :OP1 (1 2 3 ... 31 32) ) // assembles wires 1-32 and puts the constant in wire 1

(GATE :DEST 3 :OP1 1 :OP2 2 :TRUTH-TABLE #*0001 ) //AND gate on wires 1 and 2, with output in wire 3

(CONST :DEST 10 :OP1 3 ) // puts constant 3 into wire 10

(ADD :DEST 3 :OP1 1 :OP2 2 ) // adds the constants in wires 1 and 2, putting the result in 3

(INITBASE :BASE 1 ) // initializes the base pointer at 1 (0 is the global condition wire)

(CLEAR :LOCALSIZE 64 ) // clears 64 bits from the last base pointer to make room for local variables

(COPY :DEST 0 :OP1 64 :OP2 32 ) // copies 32 wires into 0-31, starting with 64

(MKPTR :DEST 10 ) // turns wire 10 into a pointer to the value it holds

(COPY-INDIR :DEST 165 :OP1 164 :OP2 32 ) // copies 32 wires pointed to by 164 into 165

(INDIR-COPY :DEST 163 :OP1 165 :OP2 32 ) // copies wires 165-197 into the location pointed by 163

(CALL :NEWBASE 100 :FNAME "alice" ) //calls "alice" setting base pointer to 100

(RET :VALUE 100 ) // returns the values in 100-131

(LABEL :STR "main" ) // symbolizes the beginning of the "main" routine

(BRANCH :CND 250 :TARG "$5" ) // branches to target "$5" on conditional wire 250
```

Figure 3.1: Examples of PCF V2 opcodes.

themselves. To perform this masking over the entire circuit, we need only to mask the input wires after they have been retrieved and to mask the output wires before they are extracted from the circuit; figure **??** shows how this is achieved. This trick gives the programmer incredibly granular control of the circuit while working at the high-level language, including the ability to specify minimal circuits using operations over single bits.

```
// With these 6 instructions, we copy 65-96 into locations 1-32.
(CONST :DEST 227 :OP1 1 )                // load const 1 into 227
(MKPTR :DEST 227 )                       // turn address 227 into a pointer; it now references 1
(CONST :DEST 228 :OP1 65 )               // load const 65 into 228
(MKPTR :DEST 228 )                       // turn address 228 into a pointer; it now references 65
(COPY-INDIR :DEST 229 :OP1 228 :OP2 32 )  // 229-260 will now contain a copy of 65-96
(INDIR-COPY :DEST 227 :OP1 229 :OP2 32 )  // the location that 227 points to will contain a copy of 229-260
```

Figure 3.2: copy-indir and indir-copy by example.

```
unsigned int input = alice(0) & 0x3F;
unsigned int somevar = myFunc(input);
...
output_alice(somevar &0x3F);
```

Figure 3.3: Input and output masks in a C program.
Only the least significant six bits of input and output are important to the PCF program. The special functions alice() and output_alice() (and corresponding bob() and output_bob()) are provided by the PCF interpreter.

3.2.1 Definitions and Algorithms

The data flow analyses used here are all adapted from Khedker, Sanyal, and Karkare [KSS09]. We begin by defining a data flow analysis, and will proceed in the following sections to describe how the equations are adapted for PCF programs.

We begin by considering an abstract program to be a sequence of operations on data, which we separate into *basic blocks*. By definition, a basic block is a maximal group of consecutive statements that are always executed together with a strictly sequential control flow between them. These basic blocks are arranged in some topological order such that when composed, the program's output is well defined. This topological ordering can be decomposed into a *control flow graph*, or CFG, which describes each basic block as well as each block's *predecessors* and *successors*. Any block b's predecessors are those which can be executed immediately before b and a block's successors can be executed immediately after b. Each edge in the CFG therefore denotes a predecessor-successor relationship, where for every edge $b_1 \to b_2$, block b_1 is the predecessor of b_2 and block b_2 is the successor of b_1. The *ancestors* of a block b are the transitive closure of b's predecessors. The *descendants* of a block b are the transitive closure of b's successors. Note that if b is located in a loop, the intersection of b's ancestors and descendants will not be empty. We assume that every CFG has two distinct nodes, *Start* and *End*, which appear at the beginning and end of a program's execution, respectively.

PCF's set of basic block operations are described by the opcodes listed in Section 3.1.2, and for simplicity of data flow analysis, we enforce only one operation per basic block, although this is not strictly necessary. In a PCF circuit, the order of the program is well defined by the output of the compiler, so most blocks have only one predecessor or successor. Blocks may have multiple predecessors only if they are labels that are target of one or more branch instructions, and each branch instruction has two successors.

Data flow values encountered over the course of a program adhere to a partial ordering over the set of possible values. This partial ordering forms a lattice with the relation $\sqsubset$, where we say x is "weaker than" y if $x \sqsubset y$; equivalently, $x \sqsubset y$ can be read "x safely estimates y." We will define the weaker-than operation in each of our analyses. Each lattice will also have a distinct greatest element and a weakest element; we will define this for each flow analysis as well. To complete the definition of a lattice, we include a confluence operation for combining flow values of blocks, which will either be "meet" $\sqcap$ or "join" $\sqcup$, where $x \sqcap y$ denotes the least upper bound of both x and y, and $x \sqcup y$ denotes the greatest lower bound of x and y in the data lattice. In our data flow analyses, the meet operation will be similar to set-intersection and the join operation will be set-union.

Each block in the CFG operates on some subset of the data that is available to it at the time of its execution. The relationship between information available before and after a block's execution is given by a set of linear simultaneous equations called *data flow equations*, defined as follows: In_n is the set of information available before a block's execution, Out_n is the set of information available after a block's execution, Gen_n is the set of information created during the execution of a block, and $Kill_n$ is the set of information which becomes invalid during a block's execution. While performing analysis, the framework will compute a flow function over the operations within each block to produce the information available after the block. Generally, this function will follow the equations $Out_n = \bigsqcup_{x \in In_n} f_n(x)$, where $f_n(x) = (x - Kill_n(x)) \cup Gen_n(x)$, and $In_n = Out_{n-1}$. By convention, the information available at the *Start* and *End* nodes are BI_{start} and BI_{end}, respectively.

The nature of some data flow problems requires us to move forward through a CFG, backward through a CFG, or in both directions on a CFG. For simplicity, our analyses do not require moving in both directions at the same time, although we do perform forward and backward data flows in sequence. In a forward data flow analysis, we will begin by analyzing the *Start* node and move successively through the successors of a node until each data flow variable converges. In a backward data flow analysis, we begin at *End* and iterate through predecessors. The order in which blocks are traversed in a program is given by the worklist algorithm for computing the maximum fixed point assignment at each program point. The textbook worklist algorithm given is reproduced in Algorithm 1; it is not changed for PCF.

We note that in lines 17 and 22 of the algorithm, the data structure used to implement the worklist has a profound impact on the algorithm's efficiency. For example, if new values are appended to the end of a trivial list at every iteration, then the analysis will begin its first pass with the first block in the circuit and proceed to the final block, after which it perform a second pass beginning with the second block, and so on until the analysis becomes an $O(n^2)$ operation, where n is the number of PCF opcodes in the circuit. We use a pairing heap to remove the minimum-index instruction at each step (maximum index for backward flows). Additionally, we keep track of the nodes already in the worklist by maintaining a set data structure (described in Section 3.2.5) to prevent adding duplicate entries to the worklist. As a result, the algorithm iterates over loops until the values within have found their maximum fixed point assignments, and inner loops converge to their assignments before outer loops. Although we iterate over loops we do not unroll them to their full depth, as we need only iterate until the variables in the loop meet their fixed point assignments.

Algorithm 1 Worklist Algorithm for Computing Data Flow [KSS09]

```
for all $u \in Points$ do
  if $u == START$ then
    $Initial_u = BI_{Start}$
  else
    if $u == END$ then
      $Initial_u = BI_{End}$
    else
      $Initial_u = \top$
    end if
  end if
  $x_u = Initial_u \sqcap (\prod_{v \in neighbors(u)} f_{v \to u}(\top))$
  if $x \sqsubset \top$ then
    add $u$ to worklist
  end if
end for
while worklist is not empty do
  Remove the first program point u from worklist
  for all $v \in neighbors(u)$ do
    $temp = x_v \sqcap f_{u \to v}(x_u)$
    if $temp \sqsubset x_v$ then
      $x_v = temp$
      add $v$ to worklist
    end if
  end for
end while
```

3.2.2 Faint Variable Analysis

Faint variable analysis is the transitive closure of dead variable analysis; a variable $x \in Var$ is defined to be *faint* at a program point u if along every path from u to End, it is either not used before being defined or is used to define a faint variable.

Faint variable analysis is a backwards data flow problem, and since it is an all-paths analysis, the confluence operator is intersection and the weakness operation is subset. Because the most aggressive optimization is to declare all variables faint, the "top" value is Var, or all variables. We use the standard textbook definitions for Gen and $Kill$, where Gen contains operations that assign or read a variable and $Kill$ contains those which use the variable. However, faint variable analysis tracks all of the faint gates in the program, while it is generally more efficient to track the live variables. Therefore, to compute our faint variable analysis, we turn the lattice upside down so that $\top$ (top of the lattice) is $\{\}$ and $\perp$ (bottom of the lattice) is Var, switch the definitions of Gen and $Kill$, and set the confluence operator to set-union. We then apply this analysis to discover all of the variables that are not faint at a program point.

The copy-indir and indir-copy opcodes require constants in their destination wires in order to dereference the location of the inputs or the outputs. In order to have all of the required information to mark dereferenced wires properly as faint (live), we must perform constant propagation analysis *before* faint variable analysis. We prefer to perform faint variable analysis as a last step in data flow analysis in order to clean up and remove all of the wires which have been deemed unnecessary by previous analyses.

3.2.3 Constant Propagation Analysis

Constant propagation analysis permits us to eliminate gates that compute wires whose values could be predicted without garbling gates and to provide important data flow information to other data flow analyses; because we consider garbling to be the bottleneck in secure computation, no circuit should compute any garbled gate online if its outputs could be predicted from a static circuit.

Constant propagation is a forward data flow problem whose lattice is actually a product of lattices for each variable in the data flow; for simplicity, we defer the discussion of the constant propagation lattice to the textbook [KSS09], except to note that non-constant wires are saved as *not-const*. The confluence operator is union, and we say that $x \sqsubset y$ if every entry in x is the same as the corresponding entry in y or *not-const*. We maintain the constants as a map from wire table index to constant value, and when merging to blocks' flow values using the confluence operator, conflicting entries become *not-const*.

The size of the constant table is a big concern for data flow analysis since it must be replicated at each basic block, and when the circuit consists of tens of thousands of operations over tens of thousands of wires, any naïve implementation of the constant table will quickly fill up the memory of the machine on which it runs. We include several optimizations intended at minimizing the constant table at each block, including compressing the entries in the table and removing constant values that are not available or necessary at the part in a program where a block exists.

3.2.4 Live Variable Analysis

By definition, a variable x is *live* at a program point u if some path from u to *End* contains a use of x which is not preceded by its definition. Because the constant table can become very large, it is useful to remove those which are not live at any point in a program, so we perform live variable analysis as a preprocessing step to constant propagation analysis and eliminate dead constants as we proceed through constant propagation. Liveness analysis for PCF circuits can be implemented according to the textbook definition with confluence operation set-union and weakness function superset.

Because we cannot track all of the pointers in the table without constant propagation analysis, our initial liveness analysis must be conservative, not eliminating any pointed-to locations in the data. We compensate for the extra weight assumed by carrying unused data by eliminating variables if we know they have passed their final uses. We construct a usage map of every variable, recording the earliest and last use of every wire in the program, so that we may eliminate extra entries in the liveness table after their last use in the map. Note that we do not use the live variable analysis for eliminating any wires from the PCF circuit, making this elimination safe to the final flow analysis.

It may seem that we could perform live variable analysis instead of our version of faint variable analysis to remove faint gates, since the two should compute the same thing on the variables. However, live variable analysis is a forward data flow analysis, and its function is to provide information about all of the variables which are still used by the program at every point. Our version of faint variable analysis is capable of doing the same thing, but we perform the backwards analysis to determine which variables are actually *important* in our program by tracking which wires contribute to the output at every program point.

3.2.5 Memory Optimizations and the Fetsko Set

Even after removing constants that are no longer live, large circuits consisting of tens of thousands of instructions quickly fill the memory of a standard machine because each block requires its own copy of the wire table. To solve the memory problems and improve both size of the flow data and speed of analysis, we introduce a special data structure and name it the Fetsko set. The Fetsko set is a functional data structure that run-length encodes an AVL-tree. It achieves the same worst-case performance as an AVL tree, but in practice it achieves excellent performance for both compression and speed in analyzing PCF circuits by leveraging the common structure of PCF data flow values. Memory in PCF circuits is composed primarily of wires grouped together in 32-bit groups (or 16-bit, 8-bit, or 4-bit groups, depending on declared data type,) whose values are likely to be related to each other. In addition, although wires holding constants may contain any integer value, most wires will have the value 0, 1, or *not-const* for unknown wires. By run-length encoding the entries in the wire table, the Fetsko set stores large numbers of wires in very little space and severely shortens the height of the look-up tree for constant values. We additionally employ Fetsko sets for liveness analysis and faint variable analysis because they improve the speed of table lookups and minimize the memory footprint of the liveness sets.

Each node of the Fetsko set contains six fields: the left tree, the right tree, the "index," the "data," the height, and the length. The index of the tree node serves as the value employed by the comparison function to order the tree. The index and the data of the node form a key-value pair comprising the node's information. For example, if the 100th wire in the circuit contains the value *not-const*, then its index and data fields would be (100,*not-const*). The length of the node stores how many following nodes contain the same data, so in the current example, a value of 5 in the length field would indicate that nodes 100-104 all contain *not-const*.

The left, right, and height fields of the Fetsko node are all the same as in an AVL tree. We use the Fetsko set to efficiently store values of the constant table as well as the liveness and faintness tables; for the latter two, all of the data fields are simply set to *true*. Inserting an individual node into the Fetsko set may be troublesome if it would border a node that shares its data type or if it would bridge the gap between two existing nodes with equivalent data fields. In the first case, the bordering node must first be removed, and then a new node is inserted with expanded ”length” field to accommodate both nodes. In the second case, the process may simply be repeated twice and the tree rebalanced afterwards. However, in such a scenario the two neighboring nodes are likely to exist in a parent-child relationship; to account for this contingency, each insert operation begins with a tree search to determine if a node join will be necessary. An insert that bridges the gap between two nodes therefore becomes a search, a remove and update, a second search, a second remove and update, and a final insert and rebalance.

Because the data structure holding the wire table is compressed, we are forced to change our weakness function to a compression-aware algorithm; this change also effects substantial improvement in the performance of the data flow analysis. Our compression-aware weakness function is simply set-subset on compressed sets; for constant propagation, this must be modified to include *not-const* as a bottom element in the data value lattice such that if checking whether $x \sqsubset y$ and $x.a \neq y.a$ but $x.a = \textit{not-const}$, then $x \sqsubset y$, but this is not true if $y.a = \textit{not-const}$ but $x.a$ is another constant.

3.3 Other Approaches to Optimization

Other systems for secure computation use a variety of approaches to circuit minimization. For example, TinyGarble [SHS+15] compiles C circuits to a hardware description language and runs standard hardware optimization tools to minimize circuits. CMBC-GC [FHK+14] unroll their loops, but also perform structural hashing, constant propagation (like PCF), and pattern-based rewriting to optimize their garbled circuits, with the goal of minimizing the total number of non-XOR gates. Our compiler favors XOR gates in the translation step from bytecode to PCF, and it is compatible with structural hashing and pattern-based rewriting, although we do not need to unroll our loops during optimization as CMBC-GC does. We note that this our approach does not perform as well as runtime systems which maintain a counter on the number of loop iterations to reduce the garbling overhead of maintaining counting variables, but for circuits with expensive computation within the loop, this overhead is small compared to the larger computation.

Chapter 4

Stable Matching

4.1 The Stable Matching Problem

Stable matching is a well studied combinatorial problem first formalized by Gale and Shapley [GS62] in the language of heterosexual, monogamous marriage. An instance of this special case of the problem consists of a set M of male suitors, a set W of female reviewers (with $|M| = |W|$ and $M \cap W = \emptyset$), and a strict preference ordering (or "ranking") of W (resp. M) for each $m \in M$ (resp. $w \in W$); we say that "m prefers w_1 to w_2" if m would rather marry w_1 than w_2. A solution to this instance consists of a one-to-one correspondence between M and W; a solution is a *stable matching* if it contains no pairs (m_1, w_1) and (m_2, w_2) such that m_1 prefers w_2 to w_1 and w_2 prefers m_1 to m_2. Gale and Shapley proved that it is *always* possible to construct such a pairing and gave an efficient algorithm to do so, described in 4.2.

In the general stable-matching problem (defined precisely in Section 5.3), the two sets of participants (*suitors* who propose matches and *reviewers* who may or may not accept the proposals) need not be of the same size, and each participant may rank only a proper subset of its potential "matches." Gusfield and Irving [GI03] provide a comprehensive overview of stable-matching algorithms.

Stable matching has seen extensive real-world use in scenarios that require the clearing of two-sided markets, including NRMP – the assignment of graduating medical students to residency programs[1], the assignment of New York City teenagers to high schools [APR05], and the assignment of Norwegian and Singaporean students to schools and universities [TST99].

[1]National Residency Matching Program, http://www.nrmp.org

4.2 The Deferred Acceptance Algorithm

The algorithm presented by Gale and Shapley to solve the stable marriage problem, popularly known as the ”Deferred Acceptance” algorithm, matches a set of *suitors* with a set of *reviewers*. As input, all participants rank each member of the opposite party from most desirable to least desirable. The algorithm progresses by permitting each unmatched suitor to propose to the reviewer he desires most among those to whom he has not yet proposed. If a reviewer has not yet received a proposal, she says ”maybe” to the proposing suitor and accepts a temporary match. If she already has a match, she compares the current offer against her temporary match, then says ”maybe” to her more desired mate and says ”no” to the other, who returns to the group of unmatched suitors. When there are no unmatched suitors left, all temporary matches are accepted as permanent and the algorithm terminates.

The algorithm provides two main guarantees. First, if the set of suitors and the set of reviewers are the same size, everyone gets matched (with the implicit assumption that a match to any partner is better than not being matched). Once a reviewer has said maybe, at no point can she be unmatched or single, since she only says no if she is already provisionally matched to a previous suitor. At the end of the protocol, no suitor or reviewer may remain unmatched, since any unmatched suitor must eventually propose to every reviewer. Second, it ensures that all the matches are stable. Let there exist two pairings $(Alice, X)$ and (Y, Bob) are where $Alice$ prefers Bob to X and Bob prefers $Alice$ to Y. Since Bob prefers $Alice$ to Y, he must have proposed to $Alice$ before he proposed to Y. At this point, $Alice$ can say either maybe or no to Bob:

- $Alice$ only says no to Bob if she prefers her current match to Bob.
- If $Alice$ says maybe to Bob, the only way in which she can be separated from Bob is if a more preferable partner proposes to her.

Thus, at the end of the protocol, if $Alice$ is not matched to Bob, she is matched with someone she prefers to Bob. The two pairings $(Alice, X)$ and (Y, Bob) cannot exist if $Alice$ prefers Bob to X and Bob prefers $Alice$ to Y.

Stable matching procedures exhibit useful properties. Roth [Rot82] showed that there will always be one stable arrangement weakly preferred by the suitors and one weakly preferred by the reviewers. The algorithm proposed by Gale and Shapley will always output a stable matching weakly preferred by the suitors; that is, each suitor will be at least as well off under the Deferred Acceptance assignment as under any other stable assignment. We say that such an assignment is *male-optimal*. (The reviewer-optimal assignment is achievable simply by flipping the suitors and reviewers.) Abdulkadiroglu et al. [APR05] and Roth [Rot82] leveraged this property in designing the New York City Match and revising the NRMP to favor students and residents.

In the worst case, the Deferred Acceptance algorithm runs in $O(n^2)$, as the maximum number of proposals is $n^2 - n + 1$ [GS62], although in practice it is usually much better than that, about $O(n \ln n)$ [Pit89]. In order for the algorithm to require the maximum number of proposals, the preference profiles of all participants must be arranged very specifically. Franklin et al. give an example of preference profiles that force the worst-case number of proposals; in order for the full $n^2 - n + 1$ proposals to be offered, preferences must be arranged so that in each round, every suitor except for one proposes to a reviewer who prefers him last among all possible remaining partners, and the final suitor knocks one suitor off of his temporary partner in each round. Gale and Shapley [GS62] also showed that it is not guaranteed that in a stable matching, any single participant gets his or her first choice of partners.

4.3 Privacy and Cheating

Agents on both sides of these markets have legitimate concerns about the privacy of their rankings and the integrity of the computation, especially in settings where preference leaks may have social or professional consequences for the participants in the future. In the terms of stable marriage, we insist that no married woman should have proof that she was anything but her husband's first choice, and likewise for a married man and his wife's first choice. However, straightforward implementations of stable-matching algorithms may leak information about participants' preferences or the history of matches made and broken on the path to a stable solution. Therefore, stable matching is a natural candidate for secure multiparty computation (SMPC). Previous works on SMPC for stable matching include those of Golle [Gol06] and Franklin et al. [FGM07a].

Prior work has shown that, in the traditional setting in which all inputs are known by all players, stable matching procedures are susceptible to manipulation by strategic players [Rot82, TST99]. Although members of the suitor group have no incentive to misrepresent their preferences [Rot82], and members of the reviewing group never benefit from misrepresenting their first choices [Rot82], a strategic player in the reviewing group may misrepresent his or her preferences to change the outcome and can sometimes switch the stable arrangement from suitor-optimal to reviewer-optimal [Rot82, TST99]. Pittel [Pit89] showed that the average rank of the reviewers' final partners is $\ln(n)$ when computing the reviewer-optimal matching and $\frac{n}{\ln(n)}$ when computing the suitor-optimal pairing. In addition, Gale and Sotomayor [GS85] showed that when partial lists are allowed, *i.e.* suitors or reviewers may mark some members of the opposite party as less desirable than remaining unmatched, the reviewers can force the reviewer-optimal matching by marking as "unacceptable" each suitor less than his or her reviewer-optimal suitor. We see that if reviewers can cheat by collaborating to precomputing the stable assignments with knowledge of the suitor preferences, they have incentive to do so.

Reviewers may also cause "faux stability" by misrepresenting their preferences; in this situation, the algorithm might output an assignment that is stable with respect to the inputs it receives but unstable with respect to the true preferences of the players. By enforcing input privacy, we leave reviewers in a position in which they will not be able to change the outcome in a predictable way. Because lying about their preferences may wind up making the outcome worse for them instead of better, simply telling the truth becomes a dominant strategy as reviewers leverage the opportunity to accept the best offer they get. Rogue reviewers may still intentionally mislead about their preferences in order to improve their results without collaborating or force an unstable matching with a beneficial aftermarket pairing, but Immorlica and Mahdian [IM05] showed that if most participants tell the truth, then honesty is indeed the dominant strategy.

4.4 Sorority Matching

We would like to apply stable matching to annual sorority recruitment at the University of Virginia, which closely resembles the matching mechanism analyzed by Mongell and Roth [MR91]. The recruitment process takes place at the beginning of each spring semester, when hundreds of female UVa undergraduates visit UVa's 16 chartered sororities. Recruitment is a collaborative process executed by sororities and "recruits" that consists of several rounds of house visitations, preference rankings, and cutoffs. In the final round of the process, each recruit writes a final preference list of three sororities to which she would like admission, and all sororities rank their remaining recruits in order of preference. The sororities and recruits submit these final preferences to a trusted third party not affiliated with the university, who computes a matching to determine all of the "bids," or offers of admission, presented by the sororities to the remaining recruits. The fact that a trusted third party is used for this computation indicates that the participants understand the highly personal nature of the input (and concomitant need for privacy safeguards) and the socially consequential nature of the output (and concomitant need to prevent byzantine manipulation). Using a PCF-based matching protocol, they could continue to maintain privacy and integrity but perform the computation without needing to trust – or pay – a third party.

Chapter 5

Secure Stable Matching

5.1 A Garbled Circuit Approach for Stable Matching

With sorority matching as motivation, we set the goal of implementing a protocol for securely computing stable marriages with hundreds of participants. Yao's garbled circuit scheme [YAO82], which is currently a popular topic yielding very fruitful research [HEKM11, CMTB13, BHKR13], is a good candidate for reasons of both efficiency and practicality. Circuit garbling employs symmetric key cryptography, which is orders of magnitude more efficient than the asymmetric operations which most multiparty protocols employ. In addition, garbled circuits can be evaluated in a constant number of rounds, which improves on the potentially large number of communication rounds incurred by private information retrieval schemes featured by other protocols, including the one by Franklin et al. [FGM07a]. Considering practicality, there do not exist good implementations of multi-party protocols at the scale of this work, involving hundreds of active participants in the computation. In addition, for such a large-scale computation, it is unlikely that every participant will behave reliably for the duration of the protocol.

Our approach performs stable matching by encoding the Gale-Shapley algorithm as a circuit that, as a first step, combines preferences that have been XOR-shared with two participants in the matching. It is important that these participants, henceforth referred to as "matching agents," not both be members of the same group in order to prevent collusion. While suitors could collude to change the stable assignment, the proposing party's dominant strategy is always to reveal its true preferences; although it may be able to alter the assignment by changing the inputs of some suitors or reviewers, any manipulation for a better personal outcome would necessarily affect the stability of the solution because the algorithm is already Pareto optimal for all of the suitors. In contrast, if both of the matching agents are reviewers, then they could collaborate to decode the suitors' preferences and then, in some cases, force the outcome of the protocol to be the reviewer-optimal stable assignment [GS85], maintaining the stability property while cheating the suitors.

After receiving the input preferences, the circuit adheres to the Gale-Shapley algorithm, albeit modified for the purposes of table look-ups. The lists of suitor preferences, reviewer preferences, suitor proposal counters, and current matches are implemented as separate arrays which must be accessed via a mux each time a specific index is desired. When considering the feasibility of circuits for stable matching, this overhead in computation has been identified as "cumbersome" by Naor et al. [NPS99]. While it is the biggest source of overhead in our computation, we show that it is still feasible to evaluate circuits for reasonably sized inputs and that it compares fairly well with the overhead encountered by other privacy-preserving techniques.

Computation and communication for our scheme are linear in the size of the circuit, which is $\theta(n^4 \lg(n))$ for a true Gale-Shapley implementation because lookups to the preference arrays are $\theta(n^2)$, lookups must be performed for each of the $\theta(n^2)$ proposals, and each player is encoded with $\lg(n)$ bits. Further modifications to the proposal algorithm based on heuristics for the number of necessary proposals reduce this complexity to $\theta(n^3 \lg(n))$. The complete garbled circuit can be evaluated in $\theta(1)$ rounds.

5.1.1 Proposed System Setup

We propose a system for many parties to compute a stable matching as follows. As the first step of the protocol, the suitors and the reviewers each select a trusted member that will be the group's matching agent. Every participant in the stable matching then sends XOR-shared inputs to the two matching agents, who subsequently use PCF to evaluate the stable matching. After circuit evaluation, the both matching agents send the results back to each participant. In the sorority rush setting, the recruits could operate a university-owned machine as their matching agent while the sororities use property of the sorority council.

We are designing a web app for stable matching in the "real world" that allows matching agents to run the secure computation for stable matching. Each matching agent downloads our web server (written in Python) and tells the other agent their own IP address. Each user registers (username, email address, and password) with their own matching agent before some designated preference submission time when the users authenticate and submit their preferences. Right before the submission time, the agents share the names of their participants (and sort them, for example alphabetically, to establish a canonical ordering). To submit preferences, each player's browser computes XOR-shares of the player's rankings and sends them to both matching agents. When the secure computation halts, the servers email their participants telling them to log in again to view the results.

Note that simply dividing a trusted third party into two entities is not a trivially generalizable approach to secure multiparty computation. We think it is reasonable for stable matching because, as discussed in Section 4.3, both groups of participants wish to maintain their own privacy and should protect their respective matching agents. Additionally, although the reviewers have incentive to collude, no suitor in the matching benefits from collaborating with the reviewers (in fact, such behavior can only hurt their outcome), so they are individually and collectively incentivized to ensure their matching agent is secure.

5.2 Alternative Solutions

5.2.1 Franklin *et al*

Golle [Gol06] and Franklin et al. [FGM07a] explored SMPC protocols for stable matching. In both works, the classic Gale-Shapley algorithm is modified to make it privacy-preserving but still computationally feasible. Golle's variant introduces semi-honest "matching authorities" and adds "dummy" men; it uses threshold homomorphic encryption and re-encryption mixnets to privately compute a stable matching. Franklin *et al.* use the same approach but devise a protocol with lower communication complexity than that of Golle [Gol06] for multi-party computation, while also providing a protocol with better performance by using only two "matching authorities" operating with xor-shared inputs. By proposing a more efficient protocol for multiparty indirect indexing, Franklin *et al.* [FGM07b] then improve on the multi-party protocol, offering the same complexity for the multiparty setting as with two matching authorities, except for a factor of the number of parties in communication complexity.

We first analyze the private stable matching by Franklin et al. [FGM07a] in which two "matching authorities" run a stable matching algorithm on xor-shared inputs. Stable matching is naturally a multi-party problem, but under reasonable trust assumptions, the bulk of the work of the computation can be done between two representative organizations whom the participants believe will not collude. This model of client-server secure computation allows most of the participants to only have to send one message to each of the organizations to participate in the protocol.

The protocol that Franklin et al. propose closely follows the traditional Gale-Shapley algorithm, slightly modified to handle shared inputs. To achieve privacy, participants XOR-share their inputs with the two matching authorities, who then operate on shared data structures. Specifically, they share four data structures: one each containing suitor and reviewer preferences, one containing the number of proposals made by each suitor, and one containing intermediate best matches. The preference arrays are of size n^2, and the bookkeeping lists are both of size n. Our protocols use the same input and model.

To compute stable assignments privately, Franklin et al employ two cryptographic tools to hide the intermediate state of the computation: a private information retrieval scheme for reading and writing the shared data structures, and Yao's garbled circuits [YAO82] for comparing reviewer preferences and incrementing the number of times a suitor has proposed. Reads and writes are accomplished using the PIR scheme of Naor and Nissim [NN01], which depends on the 1-of-W OT of Naor and Pinkas [NP01] that requires W public key operations; here, W is the size of the preference lists, or n^2, making the computation complexity $O(n^4)$. The communication complexity of the 1-of-W OT of Naor and Nissim is proportional to the size of the security parameter plus the size of a single element. The PIR scheme is run a constant number of times for each round of proposals, making the communication complexity for the entire evaluation $O(n^2)$ for information retrieval. However, the scheme requires a polylogarithmic number of rounds, making their round complexity $O(n^2 polylog(n))$. Total communication complexity for the protocol is $O(n^2 polylog(n))$ because the Yao circuits evaluated at each proposal are polylogarithmic in the number of players. We note that the protocol is complicated and no one has attempted an implementation; moreover, since most of the operations are public-key operations, our protocol will outperform it.

5.2.2 ORAM

We compare the performance of our fully garbled-gate protocol with one that uses SCORAM [WHC+14], the state of the art in ORAM for secure computation, for all of its database lookups. Since lookups to the preference tables in our implementation are $O(n^2)$, more efficient database lookups could reduce the complexity of our algorithm, potentially making incredibly large instances tractable. Using a 2^{20} byte SCORAM [WHC+14] to privately hold preference profiles would incur a cost of about 4.4 million gates. In comparison, assuming that one proposal is dominated by the two lookups into the preference tables, only our biggest test, with 500 players on each side ranking 50 partners, saw more than 4 million gates per lookup. However, even if we considered SCORAM to required only $O(1)$ rounds of communication per lookup because the size of the database is small, using SCORAM would increase the round complexity of the protocol to $O(n)$ in terms of proposals, which would take longer than the $O(1)$ rounds for evaluating a circuit.

Keller and Scholl [KS14] implement the Gale-Shapley algorithm using the SPDZ protocol of Damgård et al. [DKL+13] and oblivious arrays [LDDAM12], achieving computational complexity of $O(N^2 log^4 N)$. However, their ORAM offline phase for stable matching instances of size 10^2 is on the order of 10^7 seconds, already far worse than our implementation. Additionally, although Keller and Scholl do not report maliciously secure evaluations, our approach of using Yao's garbled circuit construction enables a trivial path to achieving malicious security. As far as we know, none of the ORAM schemes are currently capable of handling malicious adversaries due to the difficulty of compiling the ORAM step for malicious clients.

We conclude that ORAM techniques are not currently better suited for these database lookups than a mux over preference arrays for the instances we generated.

5.3 Circuits for the Deferred Acceptance Algorithm

5.3.1 Problem Statement

In our version of the problem, there is a set S of *suitors* and a set R of *reviewers*, with $S \cap R = \emptyset$ but $|S|$ not necessarily equal to $|R|$. Each suitor s ranks a subset $R_s \subseteq R$ of the reviewers, and each reviewer ranks a subset of the suitors; if $r \notin R_s$, it is understood that s considers r to be less desirable that all elements of R_s and would not agree to a match with s (and analogously for $s \notin S_r$). A solution to this more general version of the problem is a partial function $f : S \longrightarrow R$, where, in order to have $r = f(s)$, it must be the case that $r \in R_s$ and $s \in S_r$ (*i.e.*, r and s would rather be matched with each other than remain unmatched). Such a solution is stable if there are no two pairs $(s_1, r_1 = f(s_1))$ and $(s_2, r_2 = f(s_2))$ such that s_1 prefers r_2 to r_1 and r_2 prefers s_1 to s_2, and there are also no pairs (r, s) in which both r and s are unmatched but would accept each other (formally, no pairs (r, s) with $s \notin Dom(f)$, $r \notin Range(f)$, $r \in R_s$ and $s \in S_r$).

We provide a formalization of the algorithm presented in 4.2.

Formal description

We define $S = s_1, s_2, \ldots, s_n$ to be the set of suitors, and $R = r_1, r_2, \ldots, r_n$ to be the set of reviewers in the stable matching.

$M_{S,R}$ describes a stable matching as a binary relation in $S \times R$ such that $(s_i, r_j) \in M$ if $(s_i, r_j) | s_i \in S, r_j \in R$ is a pair in the stable matching.

$M'_{S,R}$ describes a set of assignments as a binary relation in $S \times R$ such that $(s_i, r_j) \in M$ if $(s_i, r_j) | s_i \in S, r_j \in R$ is a pair in the assigned matching.

$S' \subseteq S$ such that S' contains only unmatched suitors.

$(s_i, \cdot) = \{r_j | (s_i, r_i) \in M\}$

$(\cdot, r_i) = \{s_j | (s_j, r_i) \in M\}$

P describes the set of preferences for each suitor and reviewer for the members of the other party.

$p_s \in P$ describes s's preferences $[r_i, r_j, \ldots]$ such that s prefers r_i to r_j if $p_s(r_i) > p_s(r_j)$.

$p_r \in P$ describes r's preferences $[s_i, s_j, \ldots]$ such that r prefers s_i to s_j if $p_r(s_i) > p_r(s_j)$. The algorithm described in Section 4.2 is formally presented in Alg. 2.

Algorithm 2 GaleShapley(S,R,P):

```
$S' = S$
while $\exists s \in S'$ do
    $s \leftarrow S'$
    $r \leftarrow p_s | p_s(r) \geq ps(r')\ \forall r' \in R$ such that $s$ has not already chosen $r, r'$
    if $(\cdot, r) = \emptyset$ then
        $M' = M' \cup \{(s, r)\}$
        $S' = S' \backslash \{s\}$
    else
        $s' \leftarrow (\cdot, r)$
        if $p_r(s) > p_r(s')$ - that is, $r$ prefers $s$ to $s'$ then
            $M' = M' \backslash \{(s', r)\}$
            $S' = S' \cup \{s'\}$
            $M' = M' \cup \{(s, r)\}$
            $S' = S' \backslash \{s\}$
        end if
    end if
end while
return $M'$
```

5.3.2 Our Circuit Implementation

In order to perform a secure stable matching, we first design an implementation of the matching algorithm described above that does not *leak* information about the inputs. In particular, the secure computation system that we use does not hide the running time or the instructions that are executed, and thus, we must design an implementation whose running time and instruction patterns do not leak any information about the input. To do this, we use PCF to design a boolean circuit that computes the stable matching as described above.

We begin by arbitrarily ordering the suitors and reviewers and assigning them unique indices by which they will be identified. The input to the circuit consists of two matrices which represent the ordered list of preferences, *i.e.* the *preference profile* for each suitor and reviewer. Suitors format their inputs by listing reviewers in the order in which they would like to propose (*i.e.*, with reviewers sorted in decreasing order of preference). Reviewers submit preferences by listing desired suitors in the same way, with the most preferred suitor listed first and the least preferred suitor listed last.

As an example, suitor S_1 may submit an input

$$(1, 0, 2, 3)$$

to indicate that its preferences are for reviewers R_1, R_0, R_2, R_3 in order. Similarly, reviewer R_0 may submit input

$$(2, 3, 1, 0)$$

to indicate it prefers S_2, S_3, S_1, S_0 in order.

The output of the circuit consists of the match for suitors $S_1, S_2, \ldots, S_s$.

5.3.3 Gate Analysis

We define $|S|$ as the number of suitors and $|R|$ as the number of reviewers involved in the matching process. Note that there may be multiple copies of each reviewer, depending upon the *quota*, or number of suitors the reviewer is willing to accept.

We need $\lceil x \rceil$ bits to represent an integer x. The input for each suitor, which is an ordered list of $|R|$ reviewers, can therefore be described in $|R|\lceil \lg |R| \rceil$ bits. Similarly, the input for each reviewer takes $|S|\lceil \lg |S| \rceil$ bits.

Since we have $|S|$ suitors and $|R|$ reviewers, the total number of bits required is $|S||R|\lceil \lg |R| \rceil + |R||S|\lceil \lg |S| \rceil = |S||R|(\lceil \lg |R| \rceil + \lceil \lg |S| \rceil)$

In their seminal paper [GS62], Gale and Shapley prove a worst-case running time of $|S|^2 - |S| + 1$ *rounds* in their stable matching algorithm.

Within this loop, the number of gates is dominated by the need to access elements randomly from both the suitor and reviewer input arrays. The inside of the loop therefore contributes $|S||R|(\lceil \lg |R| \rceil + \lceil \lg |S| \rceil)$ gates. Overall, the circuit requires $\theta(|S|^2 \times |S||R|(\lceil \lg |R| \rceil + \lceil \lg |S| \rceil)) = \theta(|S|^3|R|(\lceil \lg |R| \rceil + \lceil \lg |S| \rceil))$ gates.

5.3.4 The "Limited" Stable Matching Adaptation

In the case of sorority matching, recruits and sororities are not permitted to rank every member of the opposite party during the final matching procedure. Rather, recruits are permitted to rank up to 3 sororities and sororities are permitted some number k that varies based on historical acceptance rates and targeted incoming class size, but is usually about 90. We simplify and generalize our implementation by assuming k does not change between suitors or between reviewers, allowing every suitor to rank up to k_s reviewers and every reviewer to rank up to k_r suitors. In the language of stable marriage, suitors and reviewers are forced to declare that after their k^{th} preferred match, they would rather remain unmarried than be bound to an undesirable partner.

Input Preferences

Because suitors and reviewers no longer submit preferences for all members of the opposing party, they truncate their preference profiles to only their k most preferred partners. For example, if suitor s were to have preference profile (2,0,1,3) with $k_s = 3$, s's inputs would be (2,0,1).

Formal Description

We introduce the following additional notation to the description given in section 5.3:

K = the set of limitations assigned to suitors and reviewers on the number of preferences they are permitted to declare, where after the k^{th} declared preference the individual prefers to remain unmatched

k_s = the limit designated for suitors on number of reviewers to rank

k_r = the limit designated for reviewers on number of suitors to rank

PC = the set of *proposal counts* for each suitor, where a proposal count indicates the number of times a suitor has submitted a proposal to a reviewer

pc_i = the number of times s_i has offered proposals to reviewers, where $0 \leq pc_i \leq k_s \forall pc \in PC$

Algorithm 3 LimitedStableMatch(S,R,P,K,PC):

```
$S' = S$
$pc_s = 0 \forall s \in S'$
while $\exists s \in S'$ do
  $s \leftarrow S'$
  if $pc_s = k_s$ then
    $S' = S' \setminus \{s\}$
  else
    $pc_s = pc_s + 1$
    $r \leftarrow p_s | p_s(r) \geq p_s(r') \; \forall r' \in R$ such that $s$ has not already chosen $r, r'$
    if $(\cdot, r) = \emptyset$ then
      $M' = M' \cup \{(s, r)\}$
      $S' = S' \setminus \{s\}$
    else
      $s' \leftarrow (\cdot, r)$
      if $p_r(s) > p_r(s')$ - that is, $r$ prefers $s$ to $s'$ then
        $M' = M' \setminus \{(s', r)\}$
        $S' = S' \cup \{s'\}$
        $M' = M' \cup \{(s, r)\}$
        $S' = S' \setminus \{s\}$
      end if
    end if
  end if
end while
return $M'$
```

Implications for Stability

While the limited matching circuit does not achieve stable outcomes in a market where all parties desire a match over no match, it does preserve stability if suitors and reviewers would truly prefer to remain unmatched than to be matched to any partner below the k^{th} preference and does preserve stability when each suitor finds a permanent match before proposing k_s times. In markets where all players prefer a partner to

Suitor preferences	R_0	R_1	R_2	R_3
S_0	0	1	2	3
S_1	3	1	2	0
S_2	0	2	3	1
S_3	2	3	1	0

Reviewer preferences	S_0	S_1	S_2	S_3
R_0	0	3	1	2
R_1	0	1	3	2
R_2	3	2	1	0
R_3	2	1	3	0

Table 5.1: Example: suitor and reviewer preference profiles

solitude, suitors are at least as well off as they would have been under the suitor-optimal stable assignment, and reviewers are at most as well off as they would have been under the suitor-optimal stable assignment.

For example, consider the preference profiles where $k_s = 2$ given by Table 5.1. Note that the suitor-optimal stable assignment for these inputs is $\{(S_0, R_0), (S_1, R_1), (S_2, R_3), (S_3, R_2)\}$, and the "limited" matching assignment is $\{(S_0, R_0), (S_1, R_3), (S_2, \emptyset), (S_3, R_2), (\emptyset, R_1)\}$. S_1 is better off under the "limited" matching procedure and R_3 is better off under the stable assignment.

5.3.5 "Limited" Circuit Gate Analysis

The number of gates required to perform the limited matching circuit differs from the basic circuit in two ways. First, the outer proposal loop is run $|S| \times k_s$ times, or until all suitors have exhausted all of their permitted proposals. Second, the submitted preferences for suitors can be described in $|S| \times k_s \times \lceil \lg |R| \rceil$ bits, and the submitted preferences for reviewers can be described in $|R| \times k_r \times \lceil \lg |S| \rceil$ bits. The inside of the proposal loop therefore contributes $|S| k_s \lceil \lg |R| \rceil + |R| k_r \lceil \lg |S| \rceil$ gates, and the loop runs $|S| \times k_s$ times, requiring a total of $\theta(|S| k_s (|S| k_s \lceil \lg |R| \rceil + |R| k_r \lceil \lg |S| \rceil))$ gates.

5.4 Running fewer proposals

Although the *worst case* instances of stable matching problems require $n^2 - n + 1$ proposals, this is not the case for most instances. In this section, we perform an ad-hoc analysis of the number of rounds needed to find a matching. We use these numbers to pick a reasonable number of proposals to make, and build circuits for these cases. Assuming that leaking the number of proposals needed to find a matching may be acceptable even though it would expose some information about generally how many proposals the men

players	mean	median	min	max	sd
10	23.94	23	10	52	6.214
50	208.4	200	97	527	50.557
100	496.6	479	241	1374	112.840
500	3353	3251	1974	7873	626.424
1000	7429	7227	4582	17160	1252.557
1500	11780	11480	7257	25760	1889.478
2000	16300	15860	10590	31400	2558.789

Table 5.2: Descriptive statistics for distributions of proposals
The average number of proposals and standard deviation increase linearly with the number of participants.
20,000 simulations for each size

must have made in order to find their mates (and therefore how desirable the final pairings are to the men), we can implement more efficient protocols by first attempting k matches and outputting an XOR-shared matching, and then running a small secure computation to see whether the result is a stable matching; if not, we can simply continue with another k proposals until the result is stable.

5.4.1 Empirical Results for Number of Proposals

We ran simulations of the standard Gale-Shapley algorithm with random inputs for party sizes between 10 and 2,000 players and found the distribution of number of proposals resembles a truncated lognormal distribution with slight right skew. Further, the average number of proposals and standard deviation grow linearly with the number of players. These results are described in Table 5.2.

To better estimate stable assignments than the “limited” circuit, we run the standard Gale-Shapley algorithm with a pre-set number of proposals. We note that “limited” circuits are generally smaller than the pre-set proposal circuits because of much cheaper preference lookups and, depending on the chosen value of K, fewer proposals. The number of proposals for a particular circuit size is set as the mean number of proposals plus six times the standard deviation for that inputs size, and we postulate that most inputs which require more proposals than this require some collusion among the players.

5.4.2 Applicability of the Simulation

The General Case

We briefly discuss whether a simulation with randomly assigned preferences is appropriate for estimating the distribution of proposals necessary when in real life, preference profiles are more likely to be drawn from some distribution arising from the relationships between people.

We know from Pittel [Pit89] that the average total rank of reviewers by the suitors, and thus the average number of proposals until a stable matching is found, is about $n\ln(n)$, which is consistent with our simulations. We argue that when inputs are drawn from some distribution over the preference profiles, the mean number of proposals is reasonably estimated by the mean when drawn from a random distribution. First, we assume that in practice, worst-case inputs like the one shown by Franklin et al. [FGM07a] which cause the long right tail of the proposal distribution are impractical because they are highly structured and require that no suitor and reviewer be mutually interested. Next, we consider extreme cases for similarity and dissimilarity. In the extreme case for similarity, when all suitors have the same preference profile, suitors propose in the same order to the reviewers, requiring $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ proposals. On the other extreme, when suitors all have different first preferences, the algorithm requires only n proposals.

For the deferred-acceptance mechanism to fail to converge quickly to a solution, the preference profiles must be *dissimilar*, since suitors must create tentative matches that are broken only down the line. When preference profiles are similar, or drawn from a distribution over which groups of suitors and reviewers are likely to have similar preferences, suitors create potential matches that are quickly broken, reducing the algorithm's search space much faster.

When we consider drawing preferences from a distribution in which some players – or groups of players – are mutually interested, we form cliques in the preference graph that approximate the simultaneous execution of multiple smaller stable matching instances. If each of these cliques were to have no crossover, *i.e.* every member of each clique ranked all of the member of his and her clique before members of any other, then for a cliques of equal size, the worst-case number of proposals would be $a\sum_{i=1}^{\frac{n}{a}} i = a\frac{\frac{n}{a}(\frac{n}{a}+1)}{2} = \frac{1}{2a}n(n+a)$ and the average would be $a\cdot\frac{n}{a}\ln(\frac{n}{a}) = n\ln(\frac{n}{a})$. Small amounts of crossover in the preference graph should not adversely affect the number of proposals, since we postulate that in practice, crossed-over preferences are likely to be rejected quickly or have the interest requited.

If we consider a case where there are a number of cliques of various sizes with maximum size b, then the worst case number of proposals would be dominated by the size of the biggest clique, and the average would be no more than $n\ln(\frac{n}{b})$. We make the critical assumption that the distribution of proposal preferences within each clique of suitors and reviews is random, which we think is more reasonable than the assumption that the entire distribution is one clique with randomly distributed preferences. We also err on the side of caution by making the number of proposals a parameter to our system, so that the size of the circuit can be set and expanded as necessary.

Sorority Matching

In the case of sorority matching, the rush process forces preference profiles to be highly structured. Indeed, in the final round of the process, recruits are only permitted to rank up to 3 sororities, but in our conversion to stable matching, each of these sororities is given 35 players in the matching that form a preference clique. Every time a new member is accepted or rejected from the sorority, up to 35 bonds must be broken and re-made as members percolate through the list, incurring the worst-case described above. However, the cliques are highly disjoint because each recruit is only allowed to rank three sororities, and from practice we know that the rush process forces a large amount of stratification among the recruits, allowing us to model the process as a stable matching with strongly connected cliques with little crossover.

5.5 Experimental Results

5.5.1 Notation

We constructed matching circuits for both the basic stable matching algorithm and our limited matching algorithm in PCF2 for a variety of circuit sizes, with and without modifications. The sizes of the matching instances are discussed in the form $|S|$x$|R|$xK_sxK_r, where $|S|$ is the number of suitors, $|R|$ is the number of reviewers, K_s is the number of reviewers whom each suitor is permitted to rank, and K_r is the number of suitors whom each reviewer is permitted to rank. In tables 5.3 and 5.4, circuits are given in NxK, where $|S| = |R| = N$ and $K_s = K_r = K$. The abbreviated notation reflects simplifications we made to the protocol for running these tests.

5.5.2 Empirical Results

For comparison, the number of gates required for a variety of feasible circuits is in Table 5.3. In addition, to easily estimate circuits for other participant sizes and numbers of proposals, we present results for the number of gates per proposal in Table 5.4.

We note that PCF has been shown capable of running circuits on the order of billions of gates [KSS12] [KasMB13], and consequentially 500x500x10x10 is close to the boundary of feasibility. We estimate the number of gates required for a complete sorority matching, which is size 560x680x90x105, by lower-bounding it with a full circuit for size 500x500x50x50. Using Table 5.4, we project the latter circuit to be about 244,819,650,000 gates, indicating sorority matching is well beyond the current scope of PCF's capability.

Circuit Size	proposals	not optimized	optimized
4×4	13	237,371	161,207
10×10	91	8,312,879	4,966,117
50×50	515	867,844,063	520,132,272
50×50	2451	4,131,719,359	2,475,800,096
100×5	500	269,144,923	175,649,776
100×10	1000	852,396,923	538,128,990
100×100	1175	7,641,520,223	4,576,531,782
200×5	1000	1,069,797,223	704,633,740
200×10	2000	3,395,801,223	2,160,526,154
500×5	2500	6,660,754,123	3,833,865,232
500×10	5000	21,172,014,123	12,262,866,201

Table 5.3: Circuit sizes in number of gates
Stable Matching instances given in $N \times K$, where $|S| = |R| = N$ and $|K_s| = |K_r| = K$, with number of proposals

Circuit Size	5 proposals	6 proposals	gates/prop
100×5	1,696,876	2,048,296	351,420
100×10	2,614,020	3,152,226	538,206
100×100	19,102,152	22,997,391	3,895,239
200×5	3,401,520	4,106,276	704,756
500×10	11,919,439	14,372,002	2,452,563
500×25	25,473,800	30,677,563	5,203,763
500×50	48,079,667	57,872,453	9,792,786

Table 5.4: Incremental gates per proposal for large optimized circuits.
Multiplying the number of incremental gates by number of desired proposals slightly overestimates size of desired circuit.

5.5.3 Runtime Analysis

We ran several of our circuits for the honest-but-curious case on c3.xlarge AWS instances with security parameter = 80. We present confidence intervals in Table 5.5. Although the larger circuits took hours to run, it is not unreasonable to use them for real matching instances when results can be computed over a weekend. We note that although we use methods described by BHKR [BHKR13], our garbling times are still considerably slower than theirs. We explain this by considering that the computation measurements include considerable time moving values through the interpreter, as our measurements do not include only encryption but all of the computation to evaluate every operation in the PCF circuit. In addition, although our circuit optimization removed a large number of gates, some garbling operations were replaced with copies, which are must but still contribute extra unnecessary work to the protocol. These copies can be replaced with copy propagation analysis, an intended direction of future work on the PCF optimizer.

		computation (s)		communication (s)	
Circuit Size	proposals	gen	eval	gen	eval
10×10	91	12.9 ± 0.2	12.3 ± 0.2	32.3 ± 0.4	33.0 ± 0.5
50×50	515	1,501 ± 6	1,417 ± 6	3,311 ± 14	3,411 ± 24
50×50	2,451	7,216 ± 73	6,779 ± 75	15,773 ± 271	16,313 ± 317
100×100	1,175	14,063 ± 694	13,138 ± 575	30,130 ± 979	31,211 ± 1,060
200×10	2,000	6,021 ± 167	5,700 ± 163	14,219 ± 203	14,585 ± 298

Table 5.5: Honest-but-curious run time evaluation
Tests were run on c3.xlarge AWS instances with security parameter 80. Confidence intervals are given for data where more than one evaluation was performed; however, for long tests few data were collected.

Chapter 6

Conclusion

We have given a garbled circuit construction for privately computing stable matching assignments, modified the algorithm for real-world variants, and analyzed its applicability to sorority matching. Stable matching is a well-motivated problem for secure multiparty computation, and because of its rich structure, poor fit for current privacy techniques, susceptibility to strategic manipulation, and high stakes of outcome, it is a good benchmark application for the field of secure computation.

Our stable matching construction highlights some of the current weaknesses in secure computation technology. Even after developing compiler optimizations and employing the latest garbling techniques, our circuits are still too big to privately compute stable matching for groups on the order of sorority matching. In the case of stable matching, the culprit is memory lookup: each preference lookup requires a MUX over the entire preference array, which is $O(n^2)$ for standard stable matching and $O(nk)$ where $k \ll n$ for our real-world variants. An oblivious algorithm and ORAM offer promising alternatives, but our efforts to produce an algorithm that would improve on Deferred Acceptance bore few fruit, and ORAM is still not efficient enough to beat circuits of our target size.

Even though we perform the simulations while sending traffic between programs through the network interface on a single machine, our simulations show that communication time now far exceeds computation time. Future research into secure computation protocols may focus on further reducing the cost of communication; since garbling techniques have become very efficient with fixed-key garbling and half-gates, it may be time to begin trading computation for communication.

The new PCF compiler shows promising results for compiling programs for secure computation, but it leaves a lot of inefficiency in the circuit design. Because PCF now simulates a stack machine, PCF circuits often waste time copying values from their permanent homes to temporary variables before operating on them and returning the values up the stack. Although the PCF optimizer removes a substantial amount of gates from the circuit and provides very granular control over boolean gates, it falls into the same trap as the compiler, producing a large number of copies. The solution is copy propagation analysis on PCF circuits as an extra step in optimization.

Circuit evaluation by the PCF interpreter has plenty of room for improvement. PCF circuits are ordered topologically, but the ordering is not usually strict. A mechanism similar to the data flow analysis could decompose the circuit into its control flow graph to evaluate independent gates simultaneously on separate threads, bringing the limiting factor of evaluation time from overall circuit size to circuit depth. Additionally, the toolchain could be improved further with a step that replaces the interpreter altogether and compiles PCF circuits directly to C programs that securely evaluate the target function.

Bibliography

[APR05] Atila Abdulkadiroglu, Parag A. Pathak, and Alvin E. Roth. The New York City high school match. *American Economic Review*, 95(2):364–367, 2005.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[CMTB13] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 289–304, Berkeley, CA, USA, 2013. USENIX Association.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: Breaking the spdz limits. In *Computer Security–ESORICS 2013*, pages 1–18. Springer, 2013.

[DNR04] Cynthia Dwork, Moni Naor, and Omer Reingold. Immunizing encryption schemes from decryption errors. In *Advances in Cryptology-EUROCRYPT 2004*, pages 342–360. Springer, 2004.

[FGM07a] Matthew Franklin, Mark Gondree, and Payman Mohassel. Improved efficiency for private stable matching. In *Topics in Cryptology – CT-RSA*, 2007.

[FGM07b] Matthew Franklin, Mark Gondree, and Payman Mohassel. Multi-party indirect indexing and applications. In *Advances in Cryptology–ASIACRYPT 2007*, pages 283–297. Springer, 2007.

[FHK$^+$14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In Albert Cohen, editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer, 2014.

[Fra91] Christopher W Fraser. A retargetable compiler for ansi c. *ACM Sigplan Notices*, 26(10):29–43, 1991.

[GI03] Dan Gusfield and Robert Irving. *The Stable Marriage Problem: Structure and Algorithms (Foundations of Computing Series)*. MIT Press, 2003.

[Gol06] Phillippe Golle. A private stable matching algorithm. In *10th International Conference on Financial Cryptography and Data Security*, 2006.

[GS62] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):pp. 9–15, 1962.

[GS85] David Gale and Marilda Sotomayor. Ms. machiavelli and the stable matching problem. *American Mathematical Monthly*, pages 261–268, 1985.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[IM05] Nicole Immorlica and Mohammad Mahdian. Marriage, honesty, and stability. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 53–62. Society for Industrial and Applied Mathematics, 2005.

[Irv85] Robert W Irving. An efficient algorithm for the ?stable roommates? problem. *Journal of Algorithms*, 6(4):577–595, 1985.

[KasMB13] Benjamin Kreuter, ahbi shelat, Benjamin Mood, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *22nd USENIX Security Symposium*, 2013.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming*, pages 486–498. Springer, 2008.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. http://eprint.iacr.org/.

[KSS09] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, pages 285–300, 2012.

[LDDAM12] John Launchbury, Iavor S Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *ACM SIGPLAN Notices*, volume 47, pages 189–200. ACM, 2012.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[MNP$^+$04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[MR91] Susan Mongell and Alvin E. Roth. Sorority rush as a two-sided matching mechanism. *The American Economic Review*, 81(3):pp. 441–464, 1991.

[NN01] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. *STOC?01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, 2001.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

[Pit89] Boris Pittel. The average number of stable matchings. *SIAM Journal on Discrete Mathematics*, 2(4):530–549, 1989.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *Advances in Cryptology–ASIACRYPT 2009*, pages 250–267. Springer, 2009.

[Rot82] Alvin E. Roth. The economics of matching: Stability and incentives. *Mathematics of Operations Research*, 7(4):pp. 617–628, 1982.

[SHS+15] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S & P*, 2015.

[TST99] C. P. Teo, J. Sethuraman, and W. P. Tan. Gale-shapley stable marriage problem revisited: strategic issues and applications. In *7th Conference on Integer Programming and Combinatorial Optimization*, 1999.

[WHC+14] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, A Shelat, and E Shi. Scoram: Oblivious ram for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[WLN+15] XS Wang, C Liu, K Nayak, Y Huang, and E Shi. Oblivm: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy (S & P)*, 2015.

[YAO82] AC YAO. Protocols for secure computation. *23rd FOCS, 1982*, 1982.

[Zah14] Samee Zahur. Obliv-c: a lightweight compiler for data-oblivious computation. In *Workshop on Applied Multi-Party Computation. Microsoft Research, Redmond*, 2014.

[ZRE14] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. Cryptology ePrint Archive, Report 2014/756, 2014. http://eprint.iacr.org/.