digital whisper
Technologic Papers

חישוב מרובה-משתתפים מאובטח ב-Rust

מאת יוראי הרצברג

הקדמה

האם שני מיליונרים, אליס ובוב, יכולים להשוות את כמות הכסף שיש להם, תחת מגבלה של פרטיות (כלומר מבלי לגלות האחד לשני דבר על כמויות הכסף)? בעיה זו היא אחת הבעיות היסודיות בתחום בקריפטוגרפיה שנקרא Secure Multi-Party Computation (SMPC). למרות שעל פניו בעיה זו נראית בלתי אפשרית, מאז שהוצגה בשנת 1982 ע"י Andrew Yao, הוצעו עבורה פתרונות רבים.

במאמר זה, נסקור את אחד הפתרונות המעניינים ביותר לבעיה זו, המבוסס על כלים קריפטוגרפיים שימושיים ביותר, הנקראים **Oblivious Transfer** ו-**Garbled Circuits**. לאחר שנלמד כיצד כלים אלו עובדים, נממש ב-Rust פרוטוקול הפותר את בעיית המיליונרים. הפרויקט הסופי מכיל כ 800 שורות קוד, ולכן לא נציג פה את כולן (הקוד המלא כאן) - נתמקד בעיקר במבני הנתונים.

בתור ידע קודם למאמר, מומלץ רקע בקריפטוגרפיה (RSA, AES) וידע בסיסי באלגברה בוליאנית.

חישוב מרובה-משתתפים מאובטח

הבעיה שהוצגה בהקדמה היא מקרה פרטי של בעיה כללית יותר: בהינתן N משתתפים, כאשר כל משתתף i מחזיק בערך x_i, ופונקציה מרובת-משתנים כלשהי f, חשבו את הערך $f(x_1,\ldots,x_N)$ בלי שאף משתתף ילמד דבר על ערכיהם של המשתתפים האחרים.

במקרה שלנו, N=2 (כי יש שני משתתפים), והפונקציה היא השוואה בוליאנית בין שני ערכים:

$f(x_1,x_2) = x_1 > x_2 \ ?\ 1 : 0$

כאשר מדברים על בעיית SMPC כלשהי, צריך להגדיר את ה-threat model: מה התוקף/תוקפים יכולים ולא יכולים לעשות. ב-SMPC, ישנם שני מודלי תקיפה עיקריים, שנבדלים בכנות של המשתתפים:

- במודל ה-semi-honest, כלומר כנה למחצה, אנחנו מניחים שהמשתתפים בפרוטוקול פועלים לפי כללי הפרוטוקול, ולא סוטים ממנו על מנת "לרמות" - לדוגמה, אליס לא יכולה לסטות מכללי הפרוטוקול על מנת לגרום לבוב להאמין שהיא יותר עשירה ממנו. מודל זה נקרא כנה למחצה (ולא כנה לגמרי), מכיוון שהמשתתפים עדיין יכולים לנתח את ההודעות באופן פסיבי, ולנסות לדלות מהן מידע.
- במודל ה-active, אין אנו מניחים הנחות בכלל על המשתתפים בפרוטוקול - כל משתתף יכול לעשות מה שהוא רוצה, ולסטות מהפרוטוקול כמה שהוא רוצה, בין אם בשביל לדלות מידע על הערכים של המשתתפים האחרים, ובין אם לשקר ולהשפיע על תוצאת הפרוטוקול.

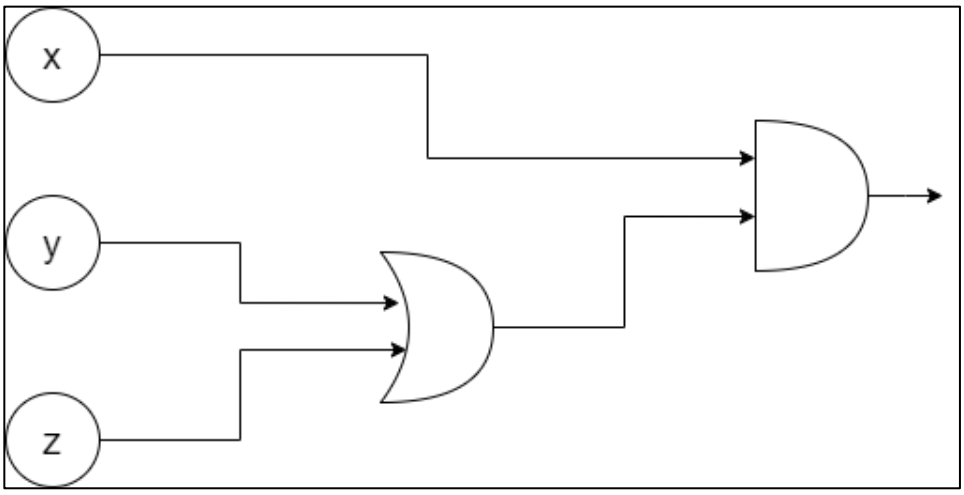
מטבע הדברים, קשה יותר להגן מפני התקפות אקטיביות, ולכן לאורך מאמר זה נניח את הנחות המודל ה-semi-honest, כלומר שהמשתתפים חייבים לעקוב אחרי כללי הפרוטוקול.

בחלק הבא, נדבר על השלב הראשון בפרוטוקול, שתפקידו לתרגם את הפונקציה שרוצים לחשב לצורה שנוח יותר לעבוד איתה.

מעגלים בוליאניים

כנראה שיצא לכם לשמוע בעבר על פונקציות בוליאניות - פונקציות שלוקחות כקלט N ערכים בוליאניים, ומוציאות כפלט M ערכים בוליאניים. לדוגמה, הפונקציה $F(x,y,z) = x(y+z)$ (כלומר $F(x,y,z) = x\ AND\ (y\ OR\ z)$):

לוקחת 3 ערכים בוליאניים ומוציאה ערך בוליאני אחד. ניתן לייצג פונקציות כאלו באמצעות DAG (גרף מכוון ללא מעגלים), כאשר הצמתים מייצגים קלטים/שערים, והקשתות (שנקראות חוטים) מייצגות חיבורים בין שערים, ממש כמו במעגל חשמלי. את הפונקציה הקודמת, לדוגמה, ניתן לייצג כך:



הקשת היוצאת משער ה-AND האחרון כוללת את הפלט של המעגל. מודל גרפי כזה נקרא מעגל בוליאני, ועליו מסתמך הפרוטוקול שלנו. מכיוון שאת רוב הדברים שניתן לבצע במחשב, ניתן לבצע באמצעות אלגברה בוליאנית, מעגלים בוליאניים מאפשרים לנו לייצג פונקציות רבות מאוד, ובפרט את פונקצית ההשוואה שאנו רוצים לחשב.

מימוש ב-Rust

כדי לייצג מעגל ב-Rust, נגדיר struct בשם Circuit, שכולל את מספר הקלטים למעגל, ואת שער הפלט בו:

```
pub struct Circuit {
    out: Node,
    /// Number of inputs to the circuit
    n: usize,
}
```

מכיוון שהקלט לכל שער תמיד מגיע דרך שני nodes אחרים, טבעי מאוד לייצג מעגל כעץ בינארי. נגדיר את Node בתור enum עם שני וריאנטים: קלט ושער. במקרה של קלט (שהוא עלה בעץ), נצטרך לאחסן את האינדקס שלו בתוך הקלט (לדוגמה הביט השלישי בקלט, הביט החמישי וכן הלאה), ובמקרה של שער, נצטרך לאחסן את פעולת השער ואת שני הבנים שלו. נעשה זאת באמצעות שמירת עמודת הפלט של השער בטבלת האמת כ-u8:

```
/// A node in the circuit
#[derive(Debug, Clone)]
pub enum Node {
    /// An input node through which the inputs to the
    Input(usize),
    /// A logic gate represented with a 4-bit integer
    Gate(u8, Box<Node>, Box<Node>),
}
```

לדוגמה, את השער OR, שטבלת האמת שלו נראית כך, נייצג כ-0b1110 (קוראים את העמודה מלמטה למעלה כי נוח יותר לבצע אינדוקס עם פעולות bitwise כך):

Input		Output
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

על מנת להריץ את ה-circuit על קלט כלשהו, כפי שנהוג כשעובדים עם עצים בינאריים, נשתמש ברקורסיה. לכל צומת בעץ, אם הצומת היא קלט, פשוט נחזיר את הקלט המתאים, ואם הצומת היא שער, נחשב רקורסיבית את הערכים של שני הבנים שלה ונפעיל את השער עליהם:

```
impl Node {
    pub fn eval(&self, input: &Vec<bool>) -> bool {
        match self {
            Node::Input(idx) => input[*idx],
            Node::Gate(op, left, right) => {
                // Index into the gate's operation based on the inputs
                let (left_val, right_val) = (left.eval(input), right.eval(input));

                (op & (1 << (2

* left_val as usize + right_val as usize))) != 0

            }
        }
    }

        ...
}
```

בשביל לבצע lookup בפעולת השער, אנו מחשבים $op \;\&\; (1 << 2 * left\ val + right_{val})$. מומלץ לוודא, שבהינתן טבלת אמת של שער מסויים (לדוגמה OR), חישוב ה-bitwise אכן מחזיר את $left\ val \vee right\ val$ לכל $0 \leq left\ val, right\ val \leq 1$.

פרוטוקול Oblivious Transfer

האלגוריתם שנממש בחלק הבא מסתמך על פרימיטיב קריפטוגרפי שנקרא *Oblivious Transfer*, או בקיצור OT. ישנם מספר סוגים של פרוטוקולי OT, אך האחד שנתרכז בו במאמר זה הוא הסוג הפשוט ביותר, שנקרא 1-2OT. בעיה זו נוגעת לשני צדדים: השולח והמקבל. השולח מחזיק בשתי הודעות m_0, m_1 (מכאן מגיע ה-2 בשם).

המקבל רוצה לקבל את אחת מההודעות של השולח (כלומר m_b עבור $0 \leq b \leq 1$), אבל מבלי שהשולח ילמד דבר על ההודעה שהמקבל בחר. דרישה זו קריטית במקרים בהן הפרטיות חשובה, כמו בבעיית המיליונרים. על מנת לפתור את בעיה זו, נשתמש בפרוטוקול שהוצע בשנת 1985 ע"י אבן, גולדרייך, ולמפל. פרוטוקול זה מבוסס על RSA, ופועל כך:

1. השולח מייצר זוג מפתחות RSA, ושולח את המפתח הפומבי למקבל (כלומר את המודולוס N ואת האקספוננט e)
2. בנוסף, השולח מייצר שתי הודעות $x_0, x_1 < N$ רנדומליות, ושולח גם אותן למקבל
3. המקבל מחליט על האינדקס b של ההודעה שהוא רוצה לקבל, ומייצר מספר רנדומלי $k < N$
4. המקבל מחשב את הערך $v = (x_b + k^e) \% N$, ושולח את התוצאה לשולח. התפקיד של צעד זה הוא בעצם לבצע blind לערכו של k, כלומר לשלוח ערך שתלוי ב-k מבלי לחשוף את k
5. באמצעות v, השולח מחשב $k_0 = (v - x_0)^d \% N, k_1 = (v - x_1)^d \% N$ כאשר d הוא המפתח הפרטי של השולח. חשוב לשים לב שעבור האינדקס b שהמקבל בחר, מתקיים $k_b = (v - x_b)^d = (x_b + k^e - x_b)^d \% N = (k^e)^d \% N = k$
6. השולח כעת מחשב $m'_0 = (m_0 + k_0) \% N, m'_1 = (m_1 + k_1) \% N$, ושולח הודעות אלו למקבל
7. מכיוון ש-$k_b = k$, המקבל יכול לחשב $m_b = m'_b - k = (m_b + k - k) = m_b$

שימו לב שבאף שלב בפרוטוקול, השולח אינו לומד דבר על האינדקס שבחר המקבל!

מימוש ב-Rust

נשמור את ה-state של השולח ב-struct הבא:

```rust
/// message m_b, without Alice finding out which message he received
pub struct ObTransferSender {
    msgs: (BigUint, BigUint),
    /// RSA keypair
    keypair: Keypair,
    /// Random messages
    xs: (BigUint, BigUint),
}
```

ה-struct מכיל את זוג ההודעות m_0 ו-m_1, ההודעות הרנדומליות x_0 ו-x_1, וזוג מפתחות RSA (למי שמתעניין בפרטי המימוש, מדובר במימוש של RSA שמימשתי ב[פוסט קודם בבלוג שלי]). לא נתעכב על אתחול ה-struct, שכן הוא כולל רק ייצור המספרים הרנדומליים x_0 ו-x_1.

באופן דומה, את המקבל נייצג כך:

```rust
/// OT from the receiver's POV
pub struct ObTransferReceiver {
    /// The xs sent by the sender
    xs: (BigUint, BigUint),
    /// Used to blind the message index
    k: BigUint,
    /// Sender's pubkey
    sender_pubkey: PublicKey,
}
```

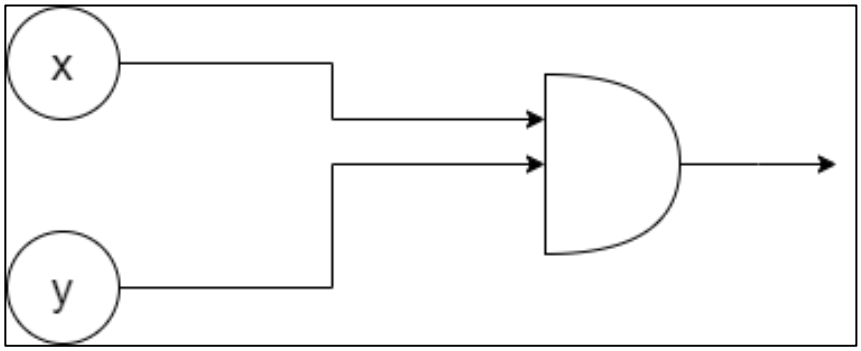
המקבל מחזיק בהודעות הרנדומליות x_0, x_1 של השולח, המספר הרנדומלי k, ומפתח ה-RSA הציבורי של השולח. המימוש של הפעולות עצמן לא מאוד מעניין, שכן מדובר רק על ייצוג הנוסחאות באלגוריתם ב-Rust, ולכן גם אותן לא אכלול כאן.

בחלק הבא, נראה כיצד ניתן להשתמש בכל הפרימיטיבים שבנינו עד כה בשביל ליצור את האלגוריתם המרכזי במאמר, המאפשר הרצה פרטית של מעגלים בוליאניים בין שני משתתפים.

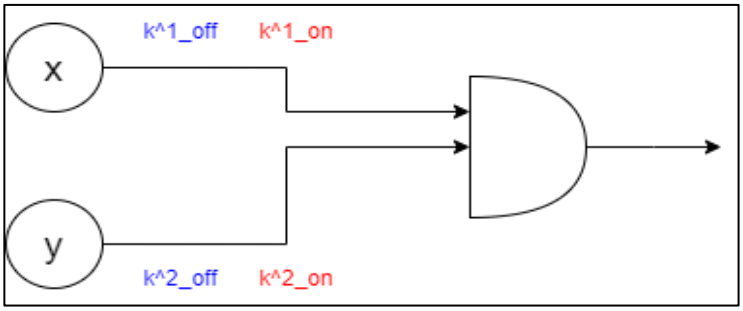
אלגוריתם Garbled Circuits

המקרה הפשוט

לאורך חלק זה, נניח שבמקום להריץ מעגל בוליאני כללי, לכל אחד משני המשתתפים יש ביט אחד (לאליס יש את x ולבוב יש את y), והם רוצים לחשב את $x \wedge y$ תחת מגבלות הפרטיות. לאחר מכן, נראה כיצד להכליל את הרעיון למעגלים כלליים. נתחיל מלהסתכל על המעגל הפשוט שלנו בצורה גרפית:



אחד מהצדדים, שאליס ובוב סיכמו עליו מראש, שנקרא ה-*garbler*, יקצה לכל אחד מהחוטים במעגל שני מפתחות: מפתח כבוי ומפתח דלוק, כמו בתמונה להלן:



כעת, לכל שער במעגל (במקרה שלנו יש רק שער AND אחד), ה-garbler מחשב את ארבעת ה-ciphertexts הבאים, אחד לכל זוג קלטים אפשרי לשער:

$$C_{00} = E(k^1{}_{off}, E(k^2{}_{off}, 0))$$

$$C_{01} = E(k^1{}_{off}, E(k^2{}_{on}, 0))$$

$$C_{10} = E(k^1{}_{on}, E(k^2{}_{off}, 0))$$

$$C_{11} = E(k^1{}_{on}, E(k^2{}_{on}, 1))$$

כאשר E היא פונקציית הצפנה כלשהי (אנחנו נשתמש ב-AES-CTR). אינטואיטיבית, מה שה-garbler עושה הוא להצפין כל פלט אפשרי של השער בהתבסס על המפתחות של חוטי הקלט. הפעולה הזו נקראת garbling, ומכאן מגיע שם האלגוריתם.

בשלב הבא, ה-garbler שולח למשתתף השני (ה-receiver) מספר הודעות:

- את ארבעת ה-ciphertexts, כלומר את $C_{00}, C_{01}, C_{10}, C_{11}$
- את המפתח המתאים לחוט ממנו מועבר ביט הקלט של ה-garbler (כלומר אם הביט של ה-garbler כבוי הוא שולח את $k^1{}_{off}$, ואחרת את $k^1{}_{on}$)

בשלב הבא, באמצעות פרימיטיב ה-OT שפיתחנו, ה-receiver מקבל את המפתח המתאים לחוט שלו (כלומר אם הביט של ה-receiver כבוי הוא יקבל את $k^2{}_{off}$). אין חשיפת מידע בשל השימוש ב-OT. נניח שהביט של ה-garbler דלוק ושהביט של ה-receiver כבוי. אם כך, ל-receiver יהיה את המידע הבא:

- $C_{00}, C_{01}, C_{10}, C_{11}$ (ה-ciphertexts)
- $k^1{}_{on}$ (חשוב להדגיש כי ה-receiver אינו יודע שמדובר ב-$k^1{}_{on}$; מבחינתו, זהו פשוט אחד מהמפתחות של ה-garbler)
- $k^2{}_{off}$

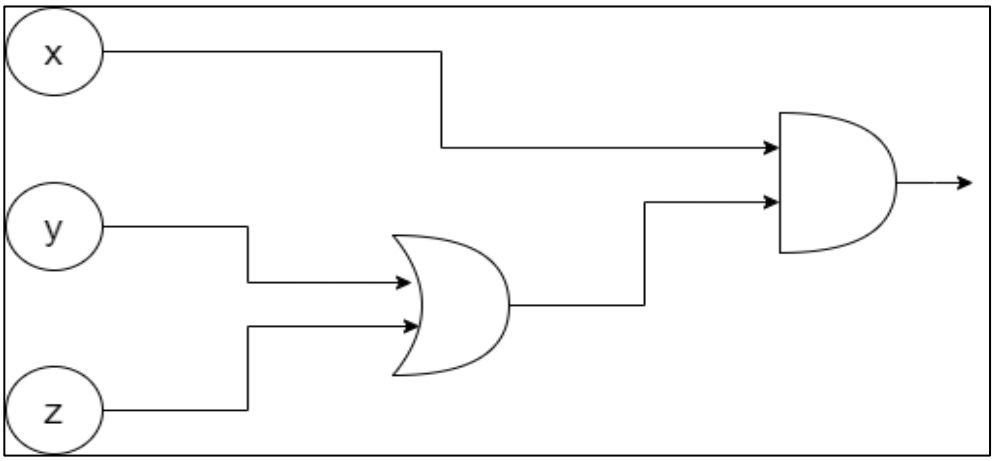
באמצעות המפתחות, ה-receiver יכול לנסות לפענח את כל אחד מארבעת ה-ciphertexts שקיבל. בגלל שרק אחד מהם הוצפן עם זוג המפתחות בהם ה-receiver מחזיק, רק ciphertext אחד יפוענח למספר $(0/1)$,

והשאר יחזירו ג'יברישׁ! במקרה זה, רק C_{10} יפוענח לתוצאה הנכונה (0), שהיא אכן התוצאה של $x \wedge y$.
לסיום, ה-receiver שולח ערך זה ל-garbler.

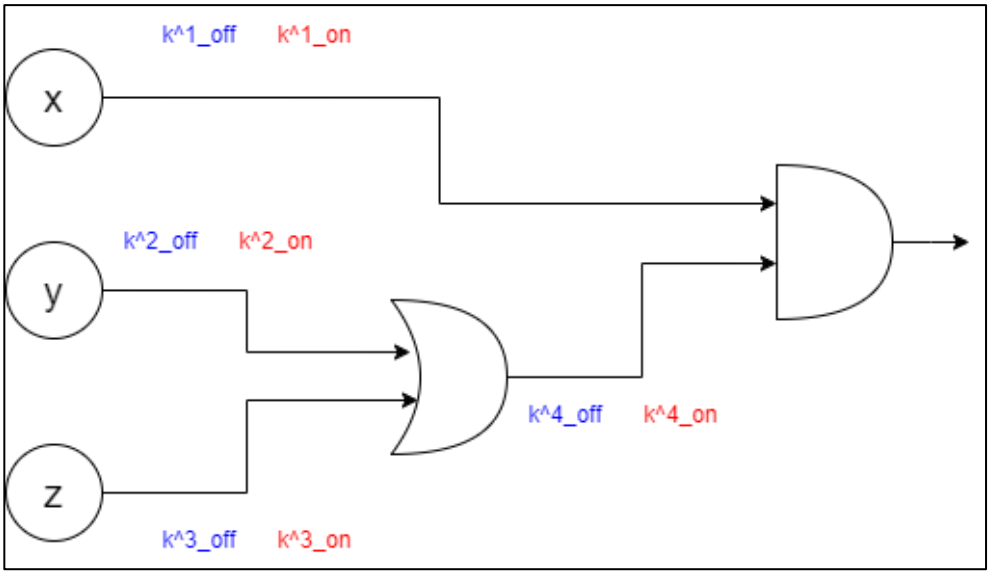
לאורך כל התהליך, אף צד לא למד מידע על הביט של הצד השני, שכן כל החישובים בוצעו על ערכים מוצפנים.
בחלק הבא, נראה כיצד ניתן להכליל את העיקרון הזה על מנת להריץ מעגלים כלליים.

מעגלים כלליים

אולי במבט ראשון נראה שיהיה קשה לבצע את ההכללה, אך התהליך אינו מסובך בהרבה מאשר במעגל הפשוט מהחלק הקודם. לאורך חלק זה, נשתמש במעגל הבא בתור דוגמה:



כמו קודם, ה-garbler ישייך לכל חוט זוג מפתחות:



ההבדל העיקרי הוא שכעת, לכל שער (חוץ משער הפלט) נצפין את המפתחות של חוט הפלט שלו (במקום להצפין 0 או 1). בשער הפלט נצפין 0 או 1 כמו מקודם, בהתאם לפלט של המעגל. לדוגמה, בשער ה-OR במעגל לעיל, ה-garbler יחשב את ה-ciphertexts הבאים:

- $C_{00} = E(k^2{}_{off}, E(k^3{}_{off}, k^4{}_{off}))$ (מצפינים את $k^4{}_{off}$ כי $0\ OR\ 0 = 0$)
- $C_{01} = E(k^2{}_{off}, E(k^3{}_{on}, k^4{}_{on}))$ (מצפינים את $k^4{}_{on}$ כי $0\ OR\ 1 = 1$)
- $C_{10} = E(k^2{}_{on}, E(k^3{}_{off}, k^4{}_{on}))$
- $C_{11} = E(k^2{}_{on}, E(k^3{}_{on}, k^4{}_{on}))$

באמצעות עקרון זה, ה-receiver יכול לקבל את המפתחות המתאימים לכל החוטים במעגל, עד שיגיע לחוט הפלט, שיוציא 0 או 1 בהתאם לפלט המתאים. ייתכן שחדי העין מבינכם שמו לב לבעיה: המפתחות לחוטים הם רנדומליים, אז איך ה-receiver ידע מבין ה-ciphertexts המפוענחים מה סתם ג'בריש ומה המפתח האמיתי? בשביל לפתור בעיה זו, בתהליך ה-garbling, לפני שנצפין את המפתחות, נוסיף להם סיומת של אפסים, וכך ה-receiver ידע שאם ciphertext מפוענח כלשהו נגמר בסיומת זו, הוא המפתח הנכון.

מימוש

בדומה למעגלים רגילים, נייצג את המעגל באמצעות עץ בינארי. נתחיל מלהגדיר struct בשם GarbledCircuit, שכולל את שער הפלט של המעגל, את מספר הקלטים בו, ואת כל החוטים שמהם מחוברים הקלטים למעגל. את השדה האחרון צריך לשמור, בשביל למנוע מצב שבו יש קלטים שמחוברים למספר שערים, עם חוטים שכוללים מפתחות שונים, מה שכמובן ימנע מהאלגוריתם לעבוד כראוי:

```rust
/// A garbled circuit from the garbler's POV
#[derive(Debug, Clone)]
pub struct GarbledCircuit {
    out: GarbledNode,
    input_wires: HashMap<usize, GarbledWire>,
    n: usize,
}
```

שוב, כמו במקרה של מעגלים רגילים, GarbledNode הוא enum שיכול להיות קלט או שער:

```rust
#[derive(Debug, Clone)]
/// Possible nodes
pub enum GarbledNode {
    Input(usize),
    Gate(Rc<RefCell<GarbledGate>>),
}
```

הפעם, לכל שער, בנוסף לבנים ופעולת השער, נצטרך לשמור מידע נוסף: את ה-ciphertexts, את החוטים שמחברים את השער לבנים שלו, ואת החוט שמחברים את השער להורה שלו.

זה ימומש כך:

```rust
#[derive(Debug, Clone)]
/// A garbled gate (from the garbler's POV, i.e.
pub struct GarbledGate {
    c_00: Option<Vec<u8>>,
    c_01: Option<Vec<u8>>,
    c_10: Option<Vec<u8>>,
    c_11: Option<Vec<u8>>,
    pub left: Option<Rc<RefCell<GarbledNode>>>,
    pub right: Option<Rc<RefCell<GarbledNode>>>,
    left_wire: Option<GarbledWire>,
    right_wire: Option<GarbledWire>,
    parent_wire: Option<GarbledWire>,
    op: Option<u8>,
}
```

כל GarbledWire מחזיק במפתח הכבוי והמפתח הדלוק, ושניהם מיוצגים כ-slice של בתים בגודל המפתח:

```rust
const KEY_SIZE: usize = 32;

#[derive(Clone, Debug)]
pub struct GarbledWire {
    on_key: [u8; KEY_SIZE],
    off_key: [u8; KEY_SIZE],
}
```

כעת, נממש את תהליך ה-garbling, כלומר המעבר ממעגל רגיל ל-GarbledCircuit. עיקר העבודה נעשית בפונקציה assign_ciphertexts, שמחשבת את ה-ciphertexts שמתאימים ל-node מסוים. ראשית כל, פונקציה זו מחשבת אילו מפתחות של חוט הפלט יש להצפין בכל שני קלטים, על בסיס טבלת האמת של השער:

```rust
impl GarbledGate {
    /// Assign ciphertexts to this gate based on its encrypted inputs
    fn assign_ciphertexts(&mut self) {
        let op = self.op.unwrap();
        // Get the bits of the operation
        let vals = ((op & 1) != 0, (op & 2) != 0, (op & 4) != 0, (op & 8) != 0);
        // Encrypt the output wire's keys
        let out_on_key = self.parent_wire.as_ref().unwrap().on_key;
        let out_off_key = self.parent_wire.as_ref().unwrap().off_key;
        // Each bit in the operation determines whether we encrypt the output wir
        let (out_00, out_01, out_10, out_11) = (
            if vals.0 { out_on_key } else { out_off_key },
            if vals.1 { out_on_key } else { out_off_key },
            if vals.2 { out_on_key } else { out_off_key },
            if vals.3 { out_on_key } else { out_off_key },
        );
            ...
        }
        ...
}
```

לדוגמה, אם vals.2 הוא true, משמעות הדבר היא שעל הקלטים T, F השער מחזיר true, ולכן יש להצפין את המפתח הדולק של חוט הפלט במקרה זה. לאחר מכן, הפונקציה מאתחלת ciphers של AES-CTR לכל אחד מהמפתחות שמחברים את ה-node לבניו (גם את AES-CTR מימשתי בפוסט קודם שלי):

```
let left_off_cipher = AesCtr::new(&self.left_wire.as_ref().unwrap().off_key);
let left_on_cipher = AesCtr::new(&self.left_wire.as_ref().unwrap().on_key);
let right_off_cipher = AesCtr::new(&self.right_wire.as_ref().unwrap().off_key);
let right_on_cipher = AesCtr::new(&self.right_wire.as_ref().unwrap().on_key);
```

לבסוף, בהתבסס על החישובים שנעשו, הפונקציה מצפינה את המפתחות של חוט הפלט (עם הסיומת של האפסים). הארגומנט השני שמועבר ל-encrypt הוא nonce. מכיוון שמדובר במימוש "צעצוע", ה-nonce הוא תמיד 0, אך כמובן שבמימוש אמיתי ה-nonces צריכים להיות רנדומליים:

```
// We append zeros to the ciphertexts so that the receiver will be able
// to distinguish between valid decryptions and gibberish
// (since the decrypted keys are, by definition, random sequences of bytes
let zeros = [0u8; KEY_SIZE];
self.c_00 = Some(left_off_cipher.encrypt(
        &right_off_cipher.encrypt([out_00, zeros].as_flattened(), 0),
        0,
));
self.c_01 = Some(left_off_cipher.encrypt(
        &right_on_cipher.encrypt([out_01, zeros].as_flattened(), 0),
        0,
));
self.c_10 = Some(left_on_cipher.encrypt(
        &right_off_cipher.encrypt([out_10, zeros].as_flattened(), 0),
        0,
));
self.c_11 = Some(left_on_cipher.encrypt(
        &right_on_cipher.encrypt([out_11, zeros].as_flattened(), 0),
        0,
));
```

בצד המקבל, נגדיר structs אנלוגיים לאלו בצד של ה-garbler, רק ללא המפתחות (שכן הם מאפשרים ללמוד את הקלט של ה-garbler):

```rust
/// From the receiver's POV, a gate is defined by its ciphertexts and its children
#[derive(Clone)]
pub struct GarbledGateRecv {
    c_00: Option<Vec<u8>>,
    c_01: Option<Vec<u8>>,
    c_10: Option<Vec<u8>>,
    c_11: Option<Vec<u8>>,
    pub left: Option<Rc<RefCell<GarbledNodeRecv>>>,
    pub right: Option<Rc<RefCell<GarbledNodeRecv>>>,
}
/// A node in the circuit can be either an input or a gate (like `Circuit` and `GarbledCircuit`)
#[derive(Clone)]
pub enum GarbledNodeRecv {
    Input(usize),
    Gate(GarbledGateRecv),
}
/// A garbled circuit from the receiver's POV
pub struct GarbledCircuitRecv {
    pub(crate) out: GarbledNodeRecv,
    pub(crate) n: usize,
}
```

בשביל להריץ מעגל שכזה, נממש את הפונקציה הרקורסיבית eval. בדומה למקבילתה על מעגלים רגילים, לכל node קלט, היא מחזירה את (המפתח של) הקלט עצמו:

```rust
impl GarbledNodeRecv {
    /// Evaluate the garbled circuit based on a vector of input keys
    pub fn eval(&self, inputs: &Vec<[u8; KEY_SIZE]>) -> [u8; KEY_SIZE] {
        match self {
            Self::Input(idx) => inputs[*idx],
            ...
        }
    }
}
```

אם ה-node הוא שער, נחשב רקורסיבית את המפתחות שיוצאים משני הבנים, ונאתחל באמצעותם ciphers של AES-CTR:

```rust
Self::Gate(gate) => {
        // Construct ciphers based on the keys coming from our left and right children
        // (this is done by recursively calling `eval` on our children)
        let left_out = gate.left.as_ref().unwrap().borrow().eval(inputs);
        let right_out = gate.right.as_ref().unwrap().borrow().eval(inputs);
        let left_cipher = AesCtr::new(&left_out);
        let right_cipher = AesCtr::new(&right_out);
        ...
}
```

לבסוף, נפענח את כל ה-ciphertexts של השער, ונבדוק איזה אחד נגמר עם סיומת האפסים:

```
// The correct key is appended with 32 zeros
let suffix = [0u8; KEY_SIZE];
// Decrypt each of this gate's ciphertexts based on the two ciphers we constructed
// Only one decryption will be valid
let d_00 =
        right_cipher.decrypt(&left_cipher.decrypt(gate.c_00.as_ref().unwrap(), 0), 0);
let d_01 =
        right_cipher.decrypt(&left_cipher.decrypt(gate.c_01.as_ref().unwrap(), 0), 0);
let d_10 =
        right_cipher.decrypt(&left_cipher.decrypt(gate.c_10.as_ref().unwrap(), 0), 0);
let d_11 =
        right_cipher.decrypt(&left_cipher.decrypt(gate.c_11.as_ref().unwrap(), 0), 0);

// Get this gate's output key by checking which decryption ends with the correct suffix
if d_00.ends_with(&suffix) {
        d_00[0..KEY_SIZE].try_into().unwrap()
} else if d_01.ends_with(&suffix) {
        d_01[0..KEY_SIZE].try_into().unwrap()
} else if d_10.ends_with(&suffix) {
        d_10[0..KEY_SIZE].try_into().unwrap()
} else {
        d_11[0..KEY_SIZE].try_into().unwrap()
}
```

בינתיים, מימשנו אלגוריתם שנותן להריץ מעגל בוליאני כלשהו בצורה פרטית. אבל עדיין אין לנו מעגל בוליאני להריץ!

מעגל השוואה

נזכור כי הפונקציה שאנו רוצים לחשב היא השוואה: $f(A,B) = A > B \ ? \ 1 \ : \ 0$, כאשר אנו מקבלים כקלט את הביטים של A ושל B. נרצה לממש מעגל שישווה את הביטים אחד-אחד, עד שיגיע לביט שונה, שבו הוא יכול לקבוע איזה מספר גדול יותר. לדוגמה, בהינתן $A=0b1010$ ו-$B=0b1011$, נרצה מעגל שישתמש בלוגיקה הבאה:

- השוואה בין ה-MSBs: מתקיים $1=1$ ולכן ממשיכים לביט הבא
- השוואה בין הביט השני משמאל: $0=0$ ולכן ממשיכים
- השוואה בין הביט השלישי משמאל: $1=1$ ולכן ממשיכים
- השוואה בין ה-LSBs: מכיוון ש-$0<1$, ניתן לקבוע כי $B>A$

בשביל לממש זאת באמצעות לוגיקה בוליאנית, ראשית נצטרך דרך לבדוק האם שני ביטים שווים. ידוע כי XOR מחזיר true רק אם שני קלטיו שונים, ולכן אם נשלול את XOR (כלומר XNOR), נוכל לקבל true רק אם שני

הקלטים שונים. אם כך, לכל $0 \leq i \leq N$, כאשר N הוא מספר הביטים ב-A ו-B, נסמן $x_i = A\ XNOR\ B$. בקוד, נממש זאת כך:

```rust
const XNOR_GATE: u8 = 0b1001u8;

/// Construct a digital comparison circuit
/// where each input is of size n bits
pub fn construct_circuit(n: usize) -> GarbledCircuit {
    let a_vals: Vec<circuit::Node> = (0..n).map(circuit::Node::Input).collect();
    let b_vals: Vec<circuit::Node> = (0..n).map(|i| circuit::Node::Input(n + i)).collect();
    let xs: Vec<circuit::Node> = (0..n).map(|i| circuit::Node::Gate(XNOR_GATE,
        Box::new(a_vals[i].clone()), Box::new(b_vals[i].clone()))
    ).collect();
```

אנחנו מגדירים את XNOR כ-0b1001 בגלל שטבלת האמת שלו נראית כך:

Input		Output
A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

אחר מכן, אנחנו יוצרים שער XNOR חדש לכל i. בחזרה למעגל, נרצה לממש את החלק של ההשוואה ביט-ביט. בהינתן שני ביטים A_i, B_i, נוכל לבדוק האם $A_i > B_i$ ע"י חישוב $A_i \wedge \neg B$, שכן הקונפיגורציה היחידה בה מתקיים $A_i > B_i$ היא $A_i = 1, B_i = 0$.

אם כך, נוכל פשוט לבדוק, האם עבור $0 \leq k \leq N$ כלשהו, מתקיים ש- k הביטים הגבוהים של A ו-B שווים, וגם שהביט ה-k של A גדול מהביט ה-k של B.

אם נתרגם זאת ללוגיקה בוליאנית, נקבל את הפונקציה הבאה (עבור מספרים בני 4 ביט; הנוסחה לקוחה מ[ויקיפדיה](ויקיפדיה)):

$$(A > B) = A_3\overline{B}_3 + x_3A_2\overline{B}_2 + x_3x_2A_1\overline{B}_1 + x_3x_2x_1A_0\overline{B}_0$$

ב-Rust, נממש זאת כך:

```rust
const XNOR_GATE: u8 = 0b1001u8;
/// $x \wedge \neg y$
/// Truth table (top to bottom):
/// F F T F
const MY_GATE: u8 = 0b0100u8;
const AND_GATE: u8 = 0b1000u8;
const OR_GATE: u8 = 0b1110u8;

let mut out: Option<circuit::Node> = None;

for i in (0..n).rev() {
    let mut cmp_hat = circuit::Node::Gate(MY_GATE, Box::new(a_vals[i].clone()), Box::new(b_vals[i].clone()));

    for x in xs.iter().take(n).skip(i+1) {
        cmp_hat = circuit::Node::Gate(AND_GATE, Box::new(cmp_hat.clone()), Box::new(x.clone()));
    }

    if out.is_some() {
        out = Some(circuit::Node::Gate(OR_GATE, Box::new(out.unwrap().clone()), Box::new(cmp_hat.clone())));
    } else {
        out = Some(cmp_hat);
    }
}

let circuit = Circuit::new(out.unwrap());

circuit.into()
```

וזה הכול! אם אתם רוצים להתעמק בחלקים בקוד שלא כיסיתי בפרויקט (לדוגמה התקשורת מעל הרשת), ניתן למצוא את הקוד המלא לפרויקט כאן.

דמו

לצערי, אני לא יכול לכלול סרטון שמראה את האלגוריתם בפעולה, ולכן אראה תמונה של כל שלב :) ניתן למצוא הדגמה חיה של האלגוריתם בערוץ היוטיוב שלי. בהדגמה זו, הסכום של כל משתתף הוא 10 ביטים, כלומר יכול לנוע מ-1 עד 1024. בהרצה הראשונה, נאזין עם ה-garbler בפורט 1337, ונכתוב שיש לנו 123 מיליון (הודעת ה-"Keypair generated" נוגעת לייצור של זוג מפתחות ה-RSA):

```
(base) yoray@BaboonZone:~/Projects/millionaire/target/release$ ./garbler 127.0.0.1 1337
How much $ do you have? (in millions): 123
Keypair generated
```

לאחר מכן, נתחבר אל ה-garbler עם ה-receiver, ונכתוב שיש לנו 456 מיליון:

```
yoray@BaboonZone:~/Projects/millionaire/target/release$ ./receiver 127.0.0.1 1337
How much $ do you have? (in millions): 456
The receiver is richer!
```

ניתן לראות שהודפס שה-receiver עשיר יותר! אותה הודעה מודפסת גם אצל ה-garbler:

```
(base) yoray@BaboonZone:~/Projects/millionaire/target/release$ ./garbler 127.0.0.1 1337
How much $ do you have? (in millions): 123
Keypair generated
The receiver is richer!
```

כעת, נהפוך את הסדר, ונכתוב של-garbler יש 888 מיליון:

```
(base) yoray@BaboonZone:~/Projects/millionaire/target/release$ ./garbler 127.0.0.1 1337
How much $ do you have? (in millions): 888
Keypair generated
```

נריץ את ה-receiver עם סכום של 887 מיליון, ונראה שהודפס שה-garbler עשיר יותר!

```
yoray@BaboonZone:~/Projects/millionaire/target/release$ ./receiver 127.0.0.1 1337
How much $ do you have? (in millions): 887
The garbler is richer!
```

סיכום

במאמר זה, מימשנו פתרון לבעיית המיליונרים ב-Rust. בשביל לעשות זאת, היינו צריכים:

- לממש API שמאפשר לעבוד עם מעגלים בוליאניים
- לבצע garbling למעגלים באמצעות אלגוריתם Garbled Circuits
- ללמוד מהו פרוטוקול OT, כיצד הוא עובד, ולממש אותו

לדעתי, זה מדהים שקיים פתרון אלגנטי כל כך לבעיה שעל פניה נראית בלתי אפשרית לפתרון. יותר מזה, בגלל יכולת ההבעה של מעגלים בוליאנים, ניתן לפתור באמצעותם גם בעיות מסובכות בהרבה. לדוגמה, שימוש מעניין הוא הרצת רשת נוירונים שלוקחת שני קלטים, בצורה פרטית (לדוגמה במצב של Federated Learning).

בשביל לעשות זאת, נצטרך להרחיב את האלגוריתם הנוכחי שלנו ולאפשר לו לעבוד עם ערכי floating point, ולחשב את ה-non-linearities ברשת (לדוגמה ReLU). גודל המעגל יגדל מהר מאוד עם גודל המודל, ולכן נצטרך למצוא דרכים לייעל את המעגל.

עוד כיוון מעניין הוא להריץ פונקציות שלוקחות יותר משני קלטים (לדוגמה $\max_{\{x_1,\ldots,x_N\}}$ עבור $N > 2$). במקרה זה, האלגוריתמים יהיו מסובכים בהרבה: על מנת שמשתתף כלשהו יוכל להריץ את המעגל, הוא יצטרך להשיג את המפתחות של כל המשתתפים - דבר שלא ניתן לעשות עם 1-2 OT כמו זה שהשתמשנו בו במאמר זה.

על הכותב

שמי יוראי הרצברג. אני סטודנט שנה רביעית למדמ"ח באוניברסיטה הפתוחה, ומתעניין מאוד ב-Security וב-AI, אני אוהב לשחק ב-CTF-ים, לעבוד על פרויקטים, ולעשות Bug Bounty.
יש לי בלוג: https://vaktibabat.github.io/ שבו אני כותב פוסטים על הנושאים הנ"ל :)

אלו הפרטים שלי:

- ה-Linkedin שלי: https://www.linkedin.com/in/yoray-herzberg-b8155621b/
- ה-GitHub שלי: https://github.com/vaktibabat
- ה-Twitter שלי: https://x.com/sag0li

מקורות

המאמר המקורי שמציג את הבעיה:

A. C. Yao, "Protocols for secure computations," 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982), Chicago, IL, USA, 1982, pp. 160-164, doi: 10.1109/SFCS.1982.38. keywords: {Protocols;Security;Algorithm design and analysis;Privacy;Voting;Databases;Stochastic processes;Cryptography;Telephony}

המאמר בו הוצג פרוטוקול ה-OT בו השתמשנו:

Shimon Even, Oded Goldreich, and Abraham Lempel. 1985. A randomized protocol for signing contracts. Commun. ACM 28, 6 (June 1985), 637-647. https://doi.org/10.1145/3812.3818

הפוסט המקורי שכתבתי על בעיה זו:

https://vaktibabat.github.io/posts/smpc_circuits/