CS 491/591 Blockchains, HW2

Adam Fasulo, Evelyn Sanchez

March 10, 2025

Question 1: Birthday Attack

(a) Collision Finding with $\mathcal{O}(2^{n/2})$ Space

For k samples from a set of size $N = 2^n$, the probability of no collision is:

$$P(\text{no collision}) = \prod_{i=0}^{k-1}\left(1 - \frac{i}{N}\right) \tag{1}$$

$$\leq e^{-\frac{k(k-1)}{2N}} \quad (\text{using } 1 - x \leq e^{-x}) \tag{2}$$

For collision probability $\geq 1/2$: $e^{-\frac{k(k-1)}{2N}} \leq \frac{1}{2} \Rightarrow k(k-1) \geq 2N\ln(2)$

For large k: $k \geq \sqrt{2N\ln(2)} \approx 1.18 \cdot 2^{n/2}$

Therefore, with $k = \lceil\sqrt{2 \cdot 2^n \cdot \ln(2)}\rceil$ hash values stored, we find a collision with probability $\geq 1/2$ in $\mathcal{O}(2^{n/2})$ time using $\mathcal{O}(2^{n/2})$ space.

(b) Space-Time Tradeoff with 2^x Space

Store 2^x hash values in a table and compute new hashes until finding a collision. Expected collision time: 2^{n-x}. Total time complexity: $T(x) = \mathcal{O}(2^x + 2^{n-x})$, minimized when $2^x = 2^{n-x} \Rightarrow x = \frac{n}{2}$, giving optimal $\mathcal{O}(2^{n/2})$ time with $\mathcal{O}(2^{n/2})$ space.

(c) Exploiting Hash Collisions in Bitcoin

Finding $x \neq y$ where $H(x) = H(y)$ enables the following double-spend attack:

1. Create transactions T_x (to merchant, normal fee) and T_y (to self, higher fee)
2. Broadcast T_x; merchant delivers goods after confirmations
3. Broadcast T_y before T_x is fully confirmed; miners prefer higher-fee T_y

Attack optimization: Target high-value transactions with precisely calibrated fee differentials to maximize profit while maintaining transaction validity. This exploits the fundamental property that only one transaction per UTXO can be confirmed.

(d) Time-Space Product Complexity $o(2^n)$

Using Hellman's Time-Memory Trade-Off with distinguished points:

- **Precomputation:** Generate m chains to distinguished points (probability 2^{-d}, expected length $t = 2^d$)
- **Online Phase:** Find distinguished point in stored table, reconstruct chain to identify collision
- **Analysis:** Space $S = \mathcal{O}(m)$, Time $T = \mathcal{O}(2^d) = \mathcal{O}(t)$

With constraint $m \cdot t^2 = 2^n$ and optimal choice $m = t = 2^{n/3}$:

$$\text{Space complexity} = \mathcal{O}(2^{n/3}) \tag{3}$$

$$\text{Time complexity} = \mathcal{O}(2^{n/3}) \tag{4}$$

Therefore, time-space product is $\mathcal{O}(2^{2n/3})$, which is $o(2^n)$.

Question 2

(a) ScriptPubKey for Square Root of 1681

Following Bitcoin ScriptPubKey checks if the supplied value is the square root of 1681:

```
OP_DUP      # Duplicate the top stack item (the supplied value)
OP_MUL      # Multiply the duplicated value by itself (square it)
1681        # Push the number 1681 onto the stack
OP_EQUAL    # Check if the squared value equals 1681
```

(b) ScriptSig to Redeem the Transaction

The following ScriptSig provides the correct square root of 1681 which is 41:

```
41          # Push the number 41 onto the stack
```

When combined with ScriptPubKey, the transaction is successfully redeemed because:

$$41^2 = 1681$$

(c) Sketch of ScriptPubKey for RSA Factoring Challenge

To create a ScriptPubKey for an RSA factoring challenge, we need to verify that the provided factors p and q multiply to N. Bitcoin Script does not natively support large number arithmetic. A theoretical implementation might look like this:

```
OP_2DUP      # Duplicate the two inputs (p and q)
OP_MULT      # Multiply p and q (Hypothetically opcode)
<N>          # Push the RSA modulus N onto the stack
OP_EQUAL     # Verify that p * q == N
```

Additional Opcodes Needed

To implement this Bitcoin Script would require:

- `OP_MULT` - Multiply two large numbers (not currently supported)
- `Arbitrary Precision` - To handle large RSA moduli
- `OP_PRIME_CHECK` (hypothetically) - Verify that p and q are prime

Partial Limitations

Bitcoin Script is intentionally limited to prevent Turing-completeness and ensure security. Supporting RSA factorization would require fundamental changes to the scripting language.

Question 3: Private Incremental Asset Acquisition Protocol

Problem Statement

Alice wants to purchase Bob's company (1%/day for 100 days at 1 BTC/1%) with privacy until completion, fairness in early termination, and no double-spending.

Solution: Hash-Locked Payment Channel with Time-Bounded Refunds

Drawing from Bitcoin's escrow and micropayment mechanisms (Lecture 3, slides 13-15), we design a private, incremental exchange protocol:

Protocol Design

Setup:

1. Alice funds 2-of-2 multisig address with 100 BTC using script:
   `OP_2 <Alice_pubkey> <Bob_pubkey> OP_2 OP_CHECKMULTISIG`
2. Bob generates secrets $\{s_i\}_{i=1}^{100}$ and sends hashes $\{H_i = \text{Hash}(s_i)\}_{i=1}^{100}$ to Alice, similar to the hash-verification technique shown in Lecture 3
3. For each day i, Bob creates and partially signs transaction TX_i that:

    - Takes the multisig output as input
    - Creates an output paying i BTC to Bob
    - Creates an output paying $(100 - i)$ BTC back to the multisig address

4. Alice creates time-locked refund transaction using `OP_CHECKLOCKTIMEVERIFY` (introduced in Lecture 3) ensuring funds return to her after 100 days

Daily Exchange ($i = 1, 2, ..., 100$):

1. Bob transfers 1% company ownership with legal documentation
2. Alice verifies documentation, signs TX_i, gives to Bob (remains unbroadcast)
3. Bob reveals s_i, Alice verifies $\text{Hash}(s_i) = H_i$
4. If continuing: transaction remains unbroadcast (following the micropayment pattern from Lecture 3)
5. If terminating: Bob broadcasts TX_i to finalize the partial purchase

Security Analysis

This construction leverages key Bitcoin script applications discussed in Lecture 3:

- **Privacy:** Utilizes the unbroadcast transaction technique (similar to micropayments in slide 14), creating only two on-chain transactions in the cooperative case
- **Fairness:** Employs the multisignature escrow pattern combined with incremental transactions to ensure proportional exchange
- **Double-Spend Prevention:** Uses 2-of-2 multisig control (`OP_CHECKMULTISIG`) to prevent unilateral fund movement, similar to the escrow example
- **Non-Cooperation Protection:** Implements time-locked refunds with `OP_CHECKLOCKTIMEVERIFY` to protect against abandonment

This solution effectively combines Bitcoin's native scripting capabilities (multisig transactions, hash verification, and time locks) to create a self-enforcing contract system without trusted third parties, directly applying the concepts from Lecture 3 "Mechanics of Bitcoin."

Question 4: Analysis of the Modified Dolev-Strong Algorithm and FLP Impossibility

The proposed modification to the Dolev-Strong flooding algorithm introduces a "round number" to each message. Processes in round r must wait to receive at least $n - t$ "round r" messages before completing that round where:

- n - Total number of processes
- t - Maximum number of faulty (crashed) processes

The goal is to determine whether this modified algorithm can solve asynchronous consensus with $t < n$ crash faults.

FLP Impossibility Theorem

The FLP impossibility theorem states that it is impossible to achieve deterministic consensus in an asynchronous system if:

1. At least one process may crash ($t \geq 1$)
2. The system is asynchronous (no bounds on message delays or process speeds)

The key insight of the FLP theorem is that the adversarial scheduler can exploit the lack of timing guarantees to prevent termination.

Why the Proposed Algorithm Fails

Key Challenges in Asynchronous Systems

In an asynchronous system:

- There is no global clock and message delays are unbounded
- A process cannot distinguish between a delayed message and a crashed process

These challenges make it difficult to guarantee the termination or progress in the presence of faults.

Adversarial Strategy

Adversarial scheduler can prevent the algorithm from terminating correctly by exploiting the following:

1. **Delayed Messages:** In round r, the scheduler can delay messages from t processes (the maximum number of faults). Non faulty processes will wait indefinitely for $n - t$ "round r" messages as they cannot distinguish between delayed messages and crashed processes.

2. **Ambiguity in Fault Detection:** Since the system is asynchronous, a process cannot determine whether another process has crashed or is just slow. This ambiguity prevents non faulty processes from making progress.
3. **Starvation:** If the scheduler continuously delays messages from t processes, non faulty processes will never receive the required $n - t$ messages for any round, resulting in the algorithm failing to terminate.

Example Scenario

Consider a system with $n = 3$ processes and $t = 1$ (one faulty process):

- Process P_1 sends a "round 1" message to P_2 and P_3
- The scheduler delays P_1's message to P_2 indefinitely
- P_2 waits for $n - t = 2$ "round 1" messages but only receives one (from P_3)
- P_2 cannot proceed to the next round because it does not receive the required number of messages

Conclusion

The proposed modification to the Dolev-Strong flooding algorithm **does not work** in an asynchronous system with $t < n$ crash faults. The FLP impossibility theorem applies directly to this case because the asynchronous nature of the system allows the adversarial scheduler to delay messages indefinitely, preventing processes from receiving the required $n - t$ messages for any round.

Question 5: Transaction Malleability Attack

Core Vulnerability

The system's reliance on transaction hashes (including signatures) creates vulnerability: signature malleability allows creation of multiple valid transaction identifiers for identical logical operations.

Attack Formalization

For transaction $Tx = (m, \sigma)$ where m is the message and σ is the signature, malleability allows deriving $\sigma' \neq \sigma$ such that:

$$\sigma' \text{ is valid for } m \tag{5}$$
$$H(m, \sigma) \neq H(m, \sigma') \tag{6}$$

This breaks the critical security property: $\text{SameEffect}(Tx_1, Tx_2) \Rightarrow H(Tx_1) = H(Tx_2)$

Execution Strategy

1. **Preparation:** Create Tx and malleable variant Tx'
2. **Selective Broadcasting:** Send each to different network subsets
3. **Race Exploitation:** After Tx is published and recorded as spent, publish Tx' which appears as a new transaction due to different hash

Advanced Exploits

- **Transaction Replacement:** Create Tx_1 to Bob, generate and publish malleable variant Tx_1', then create $Tx_2 = \{Pay\ H(Tx_1')\ to\ PK_{Mallory}, SK_{Bob}\}$
- **Protocol Disruption:** Breaks payment channels and smart contracts relying on transaction hashes

Mitigation Solutions

1. **Transaction ID Reform:** Calculate $TxID = H(\text{transaction\_data\_without\_signature})$
2. **Signature Normalization:** Enforce canonical signature format (e.g., strict DER encoding)
3. **Pre-Commitment:** Have signature cover transaction structure commitment instead of transaction itself

This attack demonstrates why transaction identification must be decoupled from malleable components to maintain consensus integrity.