Understanding PrivKit: A Complete Guide

From "What is privacy?" to "Here's how this exact code works"

Table of Contents

Part 1: For Complete Beginners

What is Privacy?

The 5-Year-Old Explanation: Imagine you have a diary. You don't want everyone to read what you wrote. That's privacy - keeping your stuff secret from others.

Real World Example: When you buy something at a store with cash, nobody knows what you bought. But when you use a credit card, the bank knows exactly what you purchased, when, and where. Privacy means having the choice to keep things to yourself.

What is Blockchain?

The 5-Year-Old Explanation: Imagine a giant notebook that everyone in the world can see. When you give your friend a sticker, you write it down: "I gave Tommy 1 sticker." Everyone can see this, and nobody can erase it. That's a blockchain.

Slightly More Advanced: A blockchain is a public ledger (record book) where every transaction is:

- **Permanent** - Once written, it can't be changed
- **Public** - Anyone can see every transaction ever made
- **Verified** - Everyone agrees the transaction happened

Solana Specifically: Solana is one particular blockchain (like how Gmail is one particular email service). It's known for being fast and cheap to use.

The Privacy Problem on Blockchain

Here's the issue: **Blockchains are TOO transparent.**

When you use normal Solana:

- Everyone can see how much money you have
- Everyone can see who you send money to
- Everyone can see every transaction you've ever made
- Your whole financial history is public forever

Why This Matters:

Imagine if everyone could see:

- Your salary (every payment from your employer)
- What you bought online (every store you paid)
- Who your friends are (everyone you sent money to)
- How rich you are (your total balance)

This isn't just uncomfortable - it's dangerous. Bad people could:

- Target rich people for theft
- Track your location based on where you spend
- Discriminate based on your financial history
- Stalk you by following your transactions

What Does PrivKit Solve?

PrivKit helps programmers build apps that protect user privacy on Solana.

It's like a toolkit. Just as a carpenter needs a hammer, saw, and nails to build a house, a programmer needs tools to build private apps. PrivKit provides those tools, pre-organized and ready to use.

The Simple Comparison:

Without PrivKit	With PrivKit
Read 100+ pages of documentation	Run 1 command
Figure out which privacy tool to use	Choose from a menu
Set up everything manually	Everything pre-configured
Hours of setup work	5 minutes to start building

Part 2: Understanding the Privacy Technologies

PrivKit supports three different privacy technologies. Think of them as three different types of locks for your diary.

Privacy Cash (The Piggy Bank with a Lock)

The 5-Year-Old Explanation

Imagine a magical piggy bank at school:

1. You put $5 in the piggy bank (nobody sees)
2. The piggy bank gives you a secret password
3. Later, you tell the piggy bank your password
4. The piggy bank gives $5 to whoever you want
5. Nobody knows it came from you!

How It Actually Works

Privacy Cash uses **Zero-Knowledge Proofs** (ZK Proofs). This is a way to prove something is true without revealing any details.

Example: Imagine proving you're old enough to enter a bar (21+) WITHOUT showing your actual birth date. You just prove "I am over 21" - nothing else.

Privacy Cash does this with money:

1. **Deposit (Shield):** You put SOL into a "privacy pool"
    - Your SOL disappears from your public wallet
    - You get a secret "note" (like a password)
2. **The Privacy Pool:** Thousands of people put money in the same pool
    - Everyone's money mixes together
    - No one can tell whose money is whose
3. **Withdraw (Unshield):** You use your secret note to get money out
    - The money can go to ANY wallet
    - No one can prove it came from you

The Code in Our Project

File: `packages/cli/templates/privacy-cash/files/src/lib/privacy/privacy-cash.ts`

```
// Deposit: Put money into the private pool
export async function deposit(
  connection: Connection,    // How we talk to Solana
  wallet: PublicKey,         // Your wallet address
  amountLamports: number     // How much (in tiny units)
): Promise<DepositResult> {
  // This gives you back a SECRET NOTE - save it!
  return {
    signature: '...',       // Proof the deposit happened
    note: '...',            // YOUR SECRET - never share this!
    commitment: '...'       // Mathematical proof for the pool
  };
}

// Withdraw: Get money out using your secret note
export async function withdraw(
  connection: Connection,
  wallet: PublicKey,
```

```
  amountLamports: number,
  recipientAddress: string,  // Where to send the money
  note?: string              // Your secret note from deposit
): Promise<WithdrawResult> {
  // The money goes to recipientAddress
  // No one can prove YOU sent it!
}
```

When to Use Privacy Cash

✅ **Perfect for:**

- Sending money privately
- Receiving payments anonymously
- Breaking the link between sender and receiver

❌ **Not ideal for:**

- Very large amounts (liquidity limits)
- Instant transactions (takes time to mix)
- Complex financial logic

Light Protocol (The Compression Magic)

The 5-Year-Old Explanation

Imagine you have 1000 photos on your phone. Each photo takes up space. Now imagine a magic camera that can squeeze 1000 photos into the space of 1 photo, and you can still see all of them perfectly. That's compression!

Light Protocol compresses your token data so it's:

- Much cheaper to store (5000x cheaper!)
- Still fully usable
- More private (compressed data is harder to read)

How It Actually Works

The Problem with Normal Tokens: On Solana, each token account costs money (rent). If you have 1000 different tokens, you pay 1000x the cost.

Light Protocol's Solution: Instead of storing each token balance as a full "account," Light Protocol:

1. Compresses all data using cryptographic techniques
2. Stores only the compressed version on-chain
3. Uses "ZK Compression" to prove the data is valid
4. Reduces costs by up to 5000x

The Privacy Benefit: Compressed data is stored differently than regular data. While not fully anonymous like Privacy Cash, it adds a layer of obscurity.

The Code in Our Project

File: `packages/cli/templates/light-protocol/files/src/lib/privacy/light-protocol.ts`

```
// Create a connection that supports compression
// IMPORTANT: Must use Helius RPC - regular Solana RPC won't work!
export function createLightConnection(heliusRpcUrl: string): Rpc {
  return createRpc(heliusRpcUrl);
}
```

```
// Create a new compressed token type
export async function createCompressedMint(
  rpc: Rpc,
  payer: Keypair,
  decimals: number = 9
): Promise<{ mint: PublicKey; signature: string }> {
  // This creates a new token that uses compression
  const { mint, transactionSignature } = await createMint(
    rpc,
    payer,
    payer.publicKey,  // Who controls the token
    decimals,         // How divisible (9 = like SOL)
    payer
  );
  return { mint, signature: transactionSignature };
}

// Transfer compressed tokens
export async function transferCompressed(
  rpc: Rpc,
  payer: Keypair,
  mint: PublicKey,
  amount: number,
  recipient: PublicKey
): Promise<string> {
  // Transfers tokens using compression
  // Much cheaper than normal transfers!
}
```

When to Use Light Protocol

✅ **Perfect for:**

- Creating new tokens cheaply
- Applications with many users (airdrops, loyalty points)
- Cost-sensitive applications
- Games with many items/tokens

❌ **Not ideal for:**

- Maximum privacy (use Privacy Cash instead)
- Compatibility with ALL Solana apps (some don't support it yet)
- Non-Helius RPC users

Arcium (The Secret Calculator)

The 5-Year-Old Explanation

Imagine you and your friends want to find out who has the most candy, but nobody wants to tell how much they have. Arcium is like a magical calculator where:

1. Everyone whispers their number to the calculator
2. The calculator figures out the answer
3. The calculator tells everyone who won
4. But the calculator NEVER reveals anyone's actual number!

How It Actually Works

Arcium uses **Multi-Party Computation (MPC)**. This allows multiple parties to compute a result together without revealing their individual inputs.

The Technical Process:

1. **Encrypt:** Your data is encrypted (scrambled) with special math
2. **Distribute:** The encrypted data goes to multiple "nodes" (computers)
3. **Compute:** The nodes do calculations ON the encrypted data
4. **Combine:** Results are combined without ever decrypting individual inputs
5. **Verify:** The result is proven correct on Solana

Real Example: Dark Pool Trading

In normal trading, if you want to buy 1 million dollars of Bitcoin:

- Everyone sees your order
- People buy before you (front-running)
- You get a worse price

With Arcium:

- Your order is encrypted
- Other orders are encrypted
- They match without revealing who ordered what
- No front-running possible!

The Code in Our Project

File: `packages/cli/templates/arcium/files/src/lib/privacy/arcium.ts`

```
// Configure connection to Arcium's MPC network
export interface ArciumConfig {
  programId: PublicKey;    // Your custom program on Solana
  clusterId?: string;       // Which MPC cluster to use
}

// Encrypted data structure
export interface EncryptedData {
  ciphertext: Uint8Array;  // The scrambled data
  nonce: Uint8Array;       // Random value for security
}

// Encrypt data before sending to MPC
export function encryptData(data: Uint8Array, nonce: Uint8Array): EncryptedData {
  // Scrambles your data so only the MPC network can compute on it
  return {
    ciphertext: data,  // In real code, this would be encrypted
    nonce
  };
}

// Submit computation to the MPC network
export async function submitComputation(
  connection: Connection,
  programId: PublicKey,
  encryptedInputs: EncryptedData[],
  instructionData: Uint8Array
```

```
): Promise<string> {
  // Sends your encrypted data to multiple computers
  // They compute together without seeing your data
}

// Wait for the computation to finish
export async function waitForFinalization(
  connection: Connection,
  programId: PublicKey,
  computationOffset: number
): Promise<ComputationResult> {
  // The MPC network finished computing
  // You get a verified result
}
```

When to Use Arcium

✅ **Perfect for:**

- Trading applications (dark pools)
- Games with hidden information (poker, battleship)
- Private auctions
- Confidential voting
- Any computation on private data

❌ **Not ideal for:**

- Simple transfers (overkill - use Privacy Cash)
- Beginners (most complex to set up)
- Speed-critical applications (MPC adds latency)

Summary: When to Use What

Scenario	Best Choice	Why
"I want to send money anonymously"	Privacy Cash	Direct private transfers
"I want to create cheap tokens for my app"	Light Protocol	5000x cost reduction
"I want to build a game with hidden cards"	Arcium	Computation on secrets
"I want to build a private DEX"	Arcium	Confidential order matching
"I want to airdrop to 1M users cheaply"	Light Protocol	Compressed account storage
"I want to receive donations anonymously"	Privacy Cash	Simple private receiving
"I want everything combined"	Full Stack	All three templates

Part 3: What PrivKit Actually Is

The Analogy

Imagine you want to build a house. You could:

1. Go to the forest and chop down trees
2. Turn trees into lumber

3. Mine iron ore for nails
4. Learn carpentry from scratch
5. Finally build your house

OR you could:

1. Buy a house kit from IKEA
2. Follow the instructions
3. Have a house in a weekend

PrivKit is the IKEA kit for building Solana privacy apps.

What It Does

PrivKit is a **CLI (Command Line Interface) tool** that creates ready-to-use project folders.

When you run:

```
npx create-solana-privacy-app my-app
```

You get a complete project with:

- All the code files pre-written
- All dependencies pre-configured
- Privacy SDK already integrated
- A working demo page
- Tests ready to run
- Deployment scripts prepared

What PrivKit Is NOT

- ❌ It's NOT a privacy service (it doesn't make anything private)
- ❌ It's NOT a blockchain (Solana is the blockchain)
- ❌ It's NOT an app (it makes apps)
- ❌ It's NOT the privacy SDKs themselves (it uses them)

Who It's For

Primary Users: Developers who want to build privacy apps on Solana

Skill Level Required:

- Must know basic programming
- Must know basic command line
- JavaScript/TypeScript helpful but not required to start

Part 4: Technical Deep Dive

Project Architecture

PrivKit is a **monorepo** (multiple packages in one repository):

```
privkit/
├── packages/
│   └── cli/                    # The CLI tool (npm package)
│       ├── src/                # Source code
│       ├── templates/          # Project templates
```

```
│       └── tests/              # CLI tests
│
├── apps/
│   └── web/                    # Marketing website
│       ├── src/
│       │   ├── app/            # Next.js pages
│       │   └── components/     # React components
│       └── public/             # Static files
│
└── package.json                # Root workspace config
```

How the CLI Works

Step 1: User Runs Command

```
npx create-solana-privacy-app my-app
```

This downloads and runs our CLI from npm.

Step 2: Entry Point

File: `packages/cli/bin/create-solana-privacy-app.js`

```
#!/usr/bin/env node
// ^^^ This "shebang" tells the OS to use Node.js

import { run } from '../dist/index.js';

run().catch((error) => {
  console.error(error);
  process.exit(1);
});
```

This file is the "front door" - it just imports and runs the main function.

Step 3: CLI Setup

File: `packages/cli/src/index.ts`

```
import { Command } from 'commander';  // CLI framework
import { input, select, password, confirm } from '@inquirer/prompts';  // Interactive prompts

export async function run(): Promise<void> {
  const program = new Command();

  program
    .name('create-solana-privacy-app')
    .description('Zero to private in one command')
    .argument('[project-name]', 'Name of the project')
    .option('-t, --template <name>', 'Template to use')
    .option('-p, --package-manager <pm>', 'Package manager', 'npm')
    // ... more options
    .action(async (projectNameArg, options) => {
      // This runs when the command executes
    });
```

```
  program.parse();  // Process the command
}
```

What Each Library Does:

- `commander` : Parses command line arguments (`--template` , `-y` , etc.)
- `@inquirer/prompts` : Shows interactive questions in terminal

Step 4: Interactive Prompts

When you run without arguments, the CLI asks questions:

```
// Ask for project name
const projectName = await input({
  message: 'What is your project name?',
  default: 'my-private-app',
  validate: (value) => {
    // Make sure the name is valid for npm
    const result = validateProjectName(value);
    return result.valid || result.error!;
  }
});

// Ask which template
const template = await select({
  message: 'Which privacy template would you like to use?',
  choices: [
    { name: 'privacy-cash     Private transfers', value: 'privacy-cash' },
    { name: 'light-protocol   ZK compression', value: 'light-protocol' },
    { name: 'arcium           MPC computation', value: 'arcium' },
    { name: 'full-stack       All combined', value: 'full-stack' }
  ]
});

// Ask for Helius API key (masked input)
const heliusKey = await password({
  message: 'Enter your Helius API key:',
  mask: '*'  // Shows **** instead of the key
});
```

Step 5: Project Generation

File: `packages/cli/src/generator.ts`

```
export async function createProject(config: ProjectConfig): Promise<void> {
  // 1. Determine paths
  const targetDir = path.resolve(process.cwd(), config.projectName);
  const templateDir = getTemplateDir(config.template);

  // 2. Check if directory already exists
  if (fs.existsSync(targetDir)) {
    throw new Error(`Directory "${config.projectName}" already exists`);
  }

  // 3. Copy and process template files
```

```
  const spinner = logger.spinner('Creating project structure');
  await processDirectory(filesDir, targetDir, {
    projectName: config.projectName,
    heliusApiKey: config.heliusKey
  });
  spinner.succeed('Created project structure');

  // 4. Create .env.local with Helius key
  const envContent = `HELIUS_API_KEY=${config.heliusKey}`;
  await fs.writeFile(envLocalPath, envContent);

  // 5. Install dependencies
  if (!config.skipInstall) {
    spinner.start('Installing dependencies');
    execSync('npm install', { cwd: targetDir });
    spinner.succeed('Installed dependencies');
  }

  // 6. Initialize git
  if (!config.skipGit) {
    execSync('git init', { cwd: targetDir });
    execSync('git add -A', { cwd: targetDir });
    execSync('git commit -m "Initial commit"', { cwd: targetDir });
  }
}
```

Step 6: Template Processing (Handlebars)

Some files need customization. We use **Handlebars** for this.

Before processing (`package.json.hbs`):

```
{
  "name": "{{projectName}}",
  "version": "0.1.0"
}
```

After processing (`package.json`):

```
{
  "name": "my-app",
  "version": "0.1.0"
}
```

The Processing Code:

```
async function processDirectory(srcDir, destDir, variables) {
  const entries = await fs.readdir(srcDir, { withFileTypes: true });

  for (const entry of entries) {
    if (entry.name.endsWith('.hbs')) {
      // This is a template file - process it
      const content = await fs.readFile(srcPath, 'utf-8');
      const template = Handlebars.compile(content);
      const processed = template(variables);  // Replace {{variables}}
```

```
      // Write without .hbs extension
      const destPath = destName.replace('.hbs', '');
      await fs.writeFile(destPath, processed);
    } else {
      // Regular file - just copy it
      await fs.copy(srcPath, destPath);
    }
  }
}
```

How Templates Work

Each template is a folder containing:

```
templates/privacy-cash/
├── template.json           # Template configuration
└── files/                  # Files to copy to new project
    ├── package.json.hbs    # Will be processed (has {{variables}})
    ├── README.md.hbs       # Will be processed
    ├── next.config.mjs     # Will be copied as-is
    ├── src/
    │   ├── app/
    │   │   └── page.tsx    # Demo UI
    │   └── lib/
    │       └── privacy/
    │           └── privacy-cash.ts  # SDK integration
    └── tests/
        └── privacy.test.ts # Ready-to-run tests
```

template.json:

```
{
  "name": "privacy-cash",
  "description": "Private transfers with Privacy Cash SDK",
  "nodeVersion": "24",
  "dependencies": {
    "privacycash": "^1.1.7"
  }
}
```

Key Files Explained

1. Logger Utility

File: `packages/cli/src/utils/logger.ts`

```
import chalk from 'chalk';  // Colors in terminal
import ora from 'ora';      // Spinning animations

export const logger = {
  // Show the PRIVKIT banner
  banner() {
    console.log(chalk.blue(`
  ██████╗ ██████╗ ██╗██╗   ██╗██╗  ██╗██╗████████╗
```

```
  ...
    `));
  },

  // Create a spinner (loading animation)
  spinner(text: string) {
    return ora(text).start();
  },

  // Show success message with next steps
  success(projectName: string, pm: string) {
    console.log(chalk.green('✓ Success!'));
    console.log(`
Next steps:
  cd ${projectName}
  ${pm} run dev
    `);
  },

  error(msg: string) {
    console.log(chalk.red('✗ Error: ' + msg));
  }
};
```

2. Validation Utility

File: `packages/cli/src/utils/validation.ts`

```
import validateNpmPackageName from 'validate-npm-package-name';
import { z } from 'zod';  // Schema validation library

// Define what a valid project name looks like
const projectNameSchema = z.string()
  .min(1, 'Project name is required')
  .max(214, 'Project name too long')
  .regex(/^[a-z0-9]/, 'Must start with lowercase letter or number');

export function validateProjectName(name: string) {
  // Check our custom rules
  const zodResult = projectNameSchema.safeParse(name);
  if (!zodResult.success) {
    return { valid: false, error: zodResult.error.message };
  }

  // Check npm's rules
  const npmResult = validateNpmPackageName(name);
  if (!npmResult.validForNewPackages) {
    return { valid: false, error: npmResult.errors?.[0] };
  }

  return { valid: true };
}
```

3. Templates Registry

File: `packages/cli/src/templates.ts`

```
export const templates = [
  {
    name: 'privacy-cash',
    description: 'Private transfers with Privacy Cash SDK',
    features: ['Zero-knowledge proofs', 'SOL & SPL token support']
  },
  {
    name: 'light-protocol',
    description: 'ZK compression with Light Protocol',
    features: ['5000x cost reduction', 'Compressed tokens']
  },
  {
    name: 'arcium',
    description: 'MPC computation with Arcium',
    features: ['Multi-party computation', 'Dark pools']
  },
  {
    name: 'full-stack',
    description: 'All privacy integrations combined',
    features: ['All three SDKs', 'Unified interface']
  }
];

export function getTemplateDir(templateName: string): string {
  // Returns: /path/to/templates/privacy-cash
  return path.join(__dirname, '..', 'templates', templateName);
}

export function checkNodeVersion(required: string) {
  const current = process.version;  // e.g., "v20.0.0"
  const currentMajor = parseInt(current.slice(1));
  const requiredMajor = parseInt(required);

  return {
    compatible: currentMajor >= requiredMajor,
    currentVersion: current
  };
}
```

Part 5: Testing & Validation

How to Test the CLI

1. Build the CLI

```
cd packages/cli
npm install
npm run build
```

2. Link It Locally

```
npm link
```

Now `create-solana-privacy-app` command works on your machine.

3. Test Creating a Project

```
# In any directory
create-solana-privacy-app test-project -t privacy-cash -y
```

4. Verify the Generated Project

```
cd test-project
npm run dev      # Should start a Next.js server
npm run build    # Should compile without errors
npm run test     # Should run tests
```

Automated Tests

File: `packages/cli/tests/` contains:

1. **CLI argument tests** - Do flags work correctly?
2. **Validation tests** - Are invalid names rejected?
3. **Generation tests** - Are files created correctly?
4. **Template tests** - Do all templates exist and have required files?

Example test:

```
import { describe, it, expect } from 'vitest';
import { validateProjectName } from '../src/utils/validation';

describe('validateProjectName', () => {
  it('accepts valid names', () => {
    expect(validateProjectName('my-app').valid).toBe(true);
    expect(validateProjectName('myapp123').valid).toBe(true);
  });

  it('rejects invalid names', () => {
    expect(validateProjectName('').valid).toBe(false);
    expect(validateProjectName('My App').valid).toBe(false);  // No spaces
    expect(validateProjectName('../hack').valid).toBe(false); // No paths
  });
});
```

What the Tests Verify

Test Category	What It Checks
Validation	Project names follow npm rules
Templates	All 4 templates exist and have required files
Generation	Handlebars variables are replaced correctly

CLI	Flags like `-t` and `-y` work as expected
Integration	Generated projects actually build and run

Part 6: Expert Reference

Cryptographic Concepts

Zero-Knowledge Proofs (ZKP)

Technical Definition: A ZKP allows a prover to convince a verifier that a statement is true without revealing any information beyond the validity of the statement.

In Privacy Cash:

- **Statement:** "I have the right to withdraw X SOL from the pool"
- **Proof:** A ZK-SNARK (Succinct Non-interactive Argument of Knowledge)
- **Revealed:** Only that the statement is true
- **Hidden:** Which deposit the withdrawal corresponds to

Key Properties:

- **Completeness:** If the statement is true, an honest prover can convince the verifier
- **Soundness:** A cheating prover cannot convince the verifier of a false statement
- **Zero-Knowledge:** The verifier learns nothing except that the statement is true

Merkle Trees

Used in: Privacy Cash for commitment storage

Structure:

```
        Root Hash
       /         \
    Hash01       Hash23
    /    \       /     \
  H0     H1    H2      H3
  |      |     |       |
Leaf0  Leaf1 Leaf2  Leaf3  (your deposits are leaves)
```

Why Useful:

- Prove a leaf exists without revealing all leaves
- Efficient verification: O(log n) instead of O(n)

Nullifiers

Used in: Privacy Cash to prevent double-spending

How It Works:

1. When you deposit, a "commitment" is added to the Merkle tree
2. When you withdraw, you reveal a "nullifier" derived from your secret
3. The nullifier is stored on-chain
4. If you try to withdraw again, the nullifier already exists = rejected

Security: The nullifier reveals NOTHING about which commitment it came from.

ZK Compression

Used in: Light Protocol

Technical Approach:

- State is stored in a compressed format using sparse Merkle trees
- Changes are proven valid using ZK proofs
- The compressed state root is stored on-chain
- Full data is stored off-chain by indexers (Helius)

Cost Reduction Math:

- Normal account: ~0.002 SOL rent
- Compressed: ~0.0000004 SOL
- Ratio: 5000x cheaper

Multi-Party Computation (MPC)

Used in: Arcium

Technical Definition: MPC allows multiple parties to jointly compute a function over their inputs while keeping those inputs private.

Arcium's Implementation:

1. **Secret Sharing:** Your input is split into "shares"
2. **Distribution:** Each share goes to a different MPC node
3. **Computation:** Nodes compute on shares without reconstructing the secret
4. **Verification:** Result is verified on Solana blockchain

Security Guarantee: As long as a threshold of nodes are honest, your input remains private.

SDK Technical Details

Privacy Cash SDK

```
// Core imports
import { deposit, withdraw, getPrivateBalance } from 'privacycash';

// Deposit creates a commitment
const result = await deposit(connection, wallet, lamports);
// result.note is YOUR SECRET - store securely!
// The note = nullifier_secret + randomness

// Withdraw uses the note to generate a proof
await withdraw(connection, wallet, lamports, recipient, note);
// Internally: generates ZK proof that note corresponds to a valid commitment
// The proof is verified on-chain

// Under the hood:
// 1. Commitment = hash(secret || nullifier || amount)
// 2. On deposit: add Commitment to Merkle tree
// 3. On withdraw: prove knowledge of secret, reveal nullifier
```

Light Protocol SDK

```
// MUST use Helius RPC
import { createRpc, Rpc } from '@lightprotocol/stateless.js';
import { createMint, mintTo, transfer } from '@lightprotocol/compressed-token';
```

```
// Helius provides the indexer for compressed state
const rpc: Rpc = createRpc('https://devnet.helius-rpc.com?api-key=KEY');

// Create compressed token
const { mint } = await createMint(rpc, payer, authority, decimals);

// Compressed transfer
await transfer(rpc, payer, mint, amount, owner, recipient);
// Internally:
// 1. Generate ZK proof of valid state transition
// 2. Update compressed state root on-chain
// 3. Helius indexes the new state
```

Arcium SDK

```
// Requires custom Anchor program
import { awaitComputationFinalization, getArciumEnv } from '@arcium-hq/client';

// 1. Set up cipher for encryption
const cipher = /* XChaCha20-Poly1305 or similar */;

// 2. Encrypt inputs
const nonce = crypto.getRandomValues(new Uint8Array(24));
const encrypted = cipher.encrypt(plaintext, nonce);

// 3. Submit to MPC via Anchor instruction
const tx = await program.methods
  .confidentialInstruction(encrypted, nonce)
  .accounts({ /* ... */ })
  .rpc();

// 4. Wait for MPC finalization
const result = await awaitComputationFinalization(
  provider,
  computationId,
  programId,
  "confirmed"
);

// 5. Decrypt result
const decrypted = cipher.decrypt(result.data, nonce);
```

Performance Considerations

SDK	Transaction Time	Cost	Throughput
Privacy Cash	5-15 seconds	~0.001 SOL	Limited by pool liquidity
Light Protocol	0.5-2 seconds	~0.00002 SOL	High (ZK compression)
Arcium	10-60 seconds	~0.005 SOL	Limited by MPC cluster

Security Model

Privacy Cash

- **Trust:** The ZK circuit (audited by Zigtur)
- **Threat:** Timing attacks if pool has low activity
- **Mitigation:** Wait for more deposits before withdrawing

Light Protocol

- **Trust:** Helius indexer (can't steal, but can censor)
- **Threat:** Indexer downtime means delayed access
- **Mitigation:** Multiple indexers planned

Arcium

- **Trust:** Threshold of MPC nodes are honest
- **Threat:** Collusion of dishonest majority
- **Mitigation:** Decentralized node selection, slashing

Quick Reference Card

CLI Commands

```
# Interactive mode
npx create-solana-privacy-app

# With all options
npx create-solana-privacy-app my-app \
  --template privacy-cash \
  --package-manager pnpm \
  --helius-key abc123 \
  --yes

# Skip steps
npx create-solana-privacy-app my-app -t light-protocol --skip-install --skip-git
```

Generated Project Commands

```
npm run dev        # Start development server (localhost:3000)
npm run build      # Build for production
npm run test       # Run tests
npm run deploy     # Deploy to Solana devnet
npm run lint       # Check code style
npm run typecheck  # Check TypeScript types
```

File Locations

What	Where
CLI entry point	`packages/cli/bin/create-solana-privacy-app.js`
Main CLI logic	`packages/cli/src/index.ts`
Project generator	`packages/cli/src/generator.ts`

Templates	`packages/cli/templates/`
Landing page	`apps/web/src/`
Privacy SDK code	`templates/*/files/src/lib/privacy/`

Glossary

Term	Simple Definition
CLI	Command Line Interface - a text-based way to run programs
SDK	Software Development Kit - pre-built code you can use
Monorepo	Multiple projects in one Git repository
Handlebars	A template language using `{{variable}}` syntax
ZKP	Zero-Knowledge Proof - prove something without revealing details
MPC	Multi-Party Computation - calculate on secrets together
RPC	Remote Procedure Call - how apps talk to Solana
Devnet	Solana's test network (free fake money)
Mainnet	Solana's real network (real money)
Lamports	Smallest unit of SOL (like cents to dollars)
Mint	A token type definition on Solana
Keypair	Public key + private key (like username + password)

This document was created to help you understand every aspect of PrivKit, from basic concepts to implementation details. If you can explain these concepts at any level - to a 5-year-old or a cryptography expert - you truly understand the project.