Hooking up with Fiat-Shamir

Ben Smyth

March 10, 2025

A zero-knowledge proof convinces us of some claim's validity without revealing specifics. For example, you can prove possession of a private key without leaking any information about your key. Zero-knowledge proofs are typically derived by application of the Fiat-Shamir transformation to sigma protocols. Unfortunately, the transformation is commonly misapplied, introducing security vulnerabilities. Herein, I present a JavaScript framework for correct transformation.

We start from a claim. For instance, I know a private key corresponding to a public key. A sigma protocol convinces us of a claim's validity; we cannot be fooled by invalid claims: A prover inputs a public statement (e.g., a public key) and a private witness (e.g., a private key) to generate a commitment, the public statement and the commitment are input by a verifier to produce a challenge, that challenge is used by the prover to generate a response, and the verifier checks whether the response demonstrates validity of the statement (Listing 1).

Sigma protocols are interactive; a prover sends a statement and a commitment to a verifier, the verifier replies with a challenge, and the prover supplies a response. The Fiat-Shamir transformation replaces the verifier's challenge with a hash over the statement and the commitment (and, optionally, some message), reducing exchange of three messages to a single proof (Listing 2).

The Fiat-Shamir transformation produces zero-knowledge proofs when the underlying sigma protocol guarantees anything derivable about a witness can be computed without interaction. As an exemplar application of our framework, let's implement the sigma protocol by Chaum et al. for demonstrating knowledge of a discrete logarithm, which may be used to prove we know a private key corresponding to a public key (Listings 3 & 4).

Securely implementing sigma protocols is as hard as implementing any comparable cryptographic primitive: Fiendishly difficult! (Just count the zero days.) I recommend the gold standard—*proven secure by design*—wherein security of sourcecode is shown to reduce to an unsolvable mathematical problem, rendering code secure assuming the math remains unsolvable. That's a tall order; I'm unaware of any such mainstream system. As an initial step, let's define a cyclic multiplicative group of prime-order using OpenSSL's implementation of Diffie-Hellman (Listings 5–7), simplifying reductions assuming OpenSSL's Diffie-Hellman can be proven secure by design.

As further exemplar applications of our framework, let's implement sigma protocols by Chaum & Pedersen for demonstrating knowledge of equality between discrete logarithms (Listing 8) and Schoenmakers for disjunctive equality between logarithms (Listings 9 & 10), which may be used to prove correctness of partial decryptions and, respectively, to prove correct ciphertext construction, the latter giving way to non-malleable ElGamal encryption over booleans.

Next steps: Provide reductions (preferably mechanised) from sourcecode to unsolvable mathematical problems, rendering implementations proven secure by design, assuming the math remains unsolvable. Building upon those reductions, demonstrate generality of techniques beyond zero-knowledge proofs by reducing code for non-malleable ElGamal to unsolvable problems. Furthermore, demonstrate generality at system level by reducing voting system sourcecode to the aforementioned unsolvable mathematical problems.

In just a year, I found zero-days in TLS implementations used by Google/Android & Oracle/OpenJDK, OpenJS Foundation/Node.js's Diffie-Hellman, and Swiss Post E-Voting, each annihilating security—clearing systems missettling trades, crypto looted, democracy stolen, could've happened. Zero-trust is the coming agenda; proven secure by design the future.

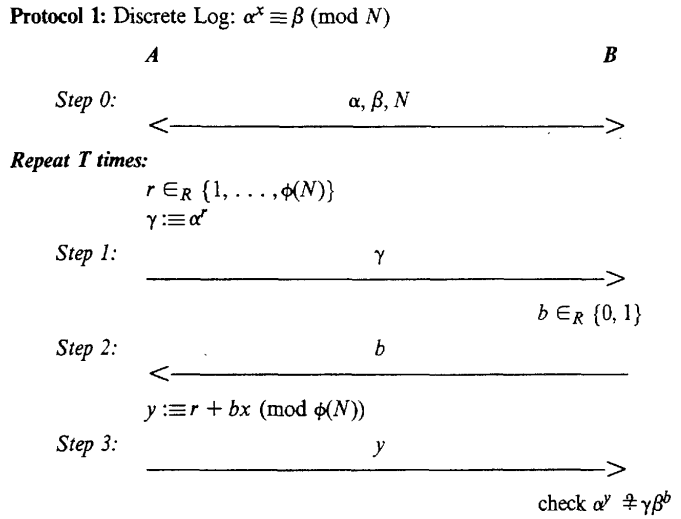
```
1  import crypto from 'crypto';
2
3  type Statement = unknown;
4  type Witness = unknown;
5
6  type Commitment = unknown;
7  type Challenge = unknown;
8  type Response = unknown;
9
10 export interface SigmaProtocol<Statement,Witness,Commitment,Challenge,Response> {
11     statement: Statement;
12
13     commit(): Commitment;
14
15     challenge?(commitment: Commitment): Challenge;
16
17     response(challenge: Challenge): Response;
18
19     verify(commitment: Commitment, challenge: Challenge, response: Response): boolean;
20 }
```

Listing 1: Sigma protocol interface comprising types for a prover's statement and their private witness (Lines 3 & 4) and the three exchanged messages, namely, a prover's commitment, a verifier's challenge, and a prover's response (Lines 6–8), a property representing a prover's statement (Line 11), methods for computing the exchanged messages, each inputting any previous message (Lines 13–17), and a final method for checking validity of a prover's statement with respect to exchanged messages (Line 19). Our challenge method is marked optional since it's not used by the Fiat-Shamir transformation.

```
22  type Proof = {commitment: Commitment, response: Response};
23
24  type Message = unknown;
25
26  export class ZeroKnowledgeProof {
27
28      private readonly sigma: SigmaProtocol<Statement,Witness,Commitment,Challenge,Response>;
29      private readonly hash: string;
30
31      constructor(sigma: SigmaProtocol<Statement,Witness,Commitment,Challenge,Response>, hash: string) {
32          this.sigma = sigma;
33          this.hash = hash;
34      }
35
36      public prove(message?: Message): Proof {
37          const commitment = this.sigma.commit();
38          const challenge = this.getChallengeHash(this.sigma.statement, commitment, message);
39          const response = this.sigma.response(challenge);
40          return {commitment, response};
41      }
42
43      public verify({commitment, response}: Proof, message?: Message): boolean {
44          const challenge = this.getChallengeHash(this.sigma.statement, commitment, message);
45          return this.sigma.verify(commitment, challenge, response);
46      }
47
48      private getChallengeHash(statement: Statement, commitment: Commitment, message?: Message): Buffer {
49          const hash = crypto.createHash(this.hash);
50          (message === undefined ? [statement, commitment] : [statement, commitment, message]).forEach(x => this.update(hash, x));
51          return hash.digest();
52      }
53
54      protected update(hash: crypto.Hash, data: unknown) : void {
55          if (typeof data === "string" || data instanceof Buffer) hash.update(data);
56          else if (data !== null && Object.values(data as any).length > 0) Object.values(data as any).forEach(x => this.update(hash, x));
57          else throw Error(`ZeroKnowledgeProof's subclass must override method update for instances of type ${typeof data}`);
58      }
59
60  }
```

Listing 2: Zero-knowledge proof derived by application of the Fiat-Shamir transformation to a sigma protocol and a hash function, comprising types capturing a prover's proof and any additional message to hash (Lines 22 & 24), instantiation of sigma protocol and hash function parameters (Lines 28–34), replacement of a verifier's challenge with a hash over a statement, a commitment, and, optionally, some message, allowing a prover to compute both their commitment & response without verifier interaction, reducing exchange of three messages to a single proof (Lines 36–41), a method for checking validity of such proofs with respect to a hash (Lines 43–46), a method for computing hashes (Lines 48–52), and a method marshalling data into a suitable format for hashing (Lines 54–58).

```typescript
import crypto from 'crypto';

import { SigmaProtocol } from './ZKP';
import { Group } from './Group';

export type Statement = {g: bigint, p: bigint, h: bigint};
export type Witness = bigint;

export type Commitment = bigint;
export type Challenge = Buffer;
export type Response = bigint;

export class SigmaProtocolDLog extends Group implements SigmaProtocol<Statement,Witness,Commitment,Challenge,Response> {

    public readonly statement: Statement;
    private readonly x!: Witness;
    protected s!: bigint;

    constructor(statement: Statement, x?: Witness) {
        super(statement.p, statement.g);
        this.statement = statement;
        if (x) this.x = x;
    }

    public commit(): Commitment {
        if (!this.x) throw new Error("A witness must be instantiated to compute a proof");
        this.s = this.random();
        return this.g(this.s) % this.prime;
    }

    public response(c: Challenge): Response {
        if (!this.s) throw new Error("Method response must be called after method commit");
        return (this.s + c.toBigint() * this.x) % this.order;
    }

    public verify(gs: Commitment, c: Challenge, r: Response): boolean {
        return this.g(r) === gs * this.exp(this.statement.h, c.toBigint()) % this.prime;
    }

}
```



Listing 3: Application of our interface implementing the sigma protocol by Chaum et al. (overlay, EuroCrypt'87), comprising instantiation of statement & (optional) witness parameter(s) and superclass initialisation (Lines 19–23), methods for computing a commitment $g^s \bmod p$ wherein s is randomly sampled (Lines 25–29), and a response $s + c \cdot x \bmod p$ (Lines 31–34) such that verification (Lines 36–38) succeeds iff $h \equiv g^x \bmod p$, i.e., when g exponentiated to the response is equivalent to the commitment multiplied by h raised to the challenge (which holds exactly when $log_g h = x$).

```
42 import { ZeroKnowledgeProof as _ZeroKnowledgeProof } from './ZKP';
43
44 export class ZeroKnowledgeProof extends _ZeroKnowledgeProof {
45     protected update(hash: crypto.Hash, x: unknown) : void {
46         if (typeof x === "bigint") hash.update(x.toBuffer());
47         else super.update(hash, x);
48     }
49 }
50
51 export class ZKPDLog extends ZeroKnowledgeProof {
52     constructor(statement: Statement, hashAlgorithm: string, witness?: Witness) {
53         super(new SigmaProtocolDLog(statement, witness), hashAlgorithm);
54     }
55 }
```

Listing 4: Application of our framework deriving a zero-knowledge proof system using the Fiat-Shamir transformation—super easy, huh?! All you've got to do is extend method update to handle any new types (Lines 45–48) and instantiate (Lines 52–54). (Whilst two classes aren't necessary, we separate for reuse of the former.)

```
1  import crypto from 'crypto';
2
3  declare global { interface Buffer {  toBigint(): bigint; }
4                   interface BigInt {  toBuffer(): Buffer; }
5                   interface BigInt {  mod(n: bigint): bigint; } }
6
7  Buffer.prototype.toBigint = function(this: Buffer): bigint {
8      return BigInt(`0x${this.toString('hex')}`);
9  }
10
11 BigInt.prototype.toBuffer = function(this: bigint): Buffer {
12     /* RTFM: "Data truncation may occur when decoding strings that do not exclusively
13        consist of an even number of hexadecimal characters," painful to debug...
14        Source: https://github.com/nodejs/node/blob/main/doc/api/buffer.md */
15     let s = this.toString(16);
16     if (s.length % 2 !== 0) s = "0" + s;
17     return Buffer.from(s, 'hex');
18 }
19
20 BigInt.prototype.mod = function(this: bigint, n: bigint): bigint {
21     /* RTFM: "The remainder (%) operator returns the remainder left over when one operand is
22        divided by a second operand. It always takes the sign of the dividend," perplexing...
23        Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Remainder */
24     return ((this%n)+n)%n;
25 }
```

Listing 5: Buffer to bigint conversions for mathematically operations on Diffie-Hellman inputs/outputs.

```
30  class _Group {
32      public readonly _prime: Buffer;
33      public readonly _generator: Buffer;
34      private readonly dh: crypto.DiffieHellman;
44      constructor(prime: Buffer, generator: Buffer) {
45          if (crypto.createDiffieHellman(prime, generator).verifyError !== 0)
46              throw new Error("Not a prime, not a safe prime, generator unsuitable, or generator suitability cannot be checked");
47
48          this._prime = prime;
49          this._generator = generator;
50          this.dh = crypto.createDiffieHellman(prime, generator);
51      }
52
53      public exp(base: Buffer, exp: Buffer): Buffer {
54          this.dh.setPrivateKey(exp);
55          return this.dh.computeSecret(base);
56      }
57
58      public random(): Buffer {
59          //Here be dragons: Use fresh Diffie-Hellman instance to avoid Node.js zero-day CVE-2023-30590; function generateKeys() "only generates a private
60          //key if none has been set" (https://nodejs.org/en/blog/vulnerability/june-2023-security-releases)
61          const dh = crypto.createDiffieHellman(this._prime, this._generator)
62          dh.generateKeys();
63          return dh.getPrivateKey();
64      }
65
66  }
```

Listing 6: Exploiting OpenSSL's implementation of Diffie-Hellman for exponentiation of a base to an exponent (Lines 53–56) and for sampling random bits (Lines 58–64).

```
72  export class Group extends _Group {
74      public readonly prime: bigint;
75      public readonly generator: bigint;
76      public readonly order: bigint;
86      constructor(prime: bigint, generator: bigint) {
87          super(prime.toBuffer(), generator.toBuffer());
88          this.prime = prime;
89          this.generator = generator;
90          this.order = (prime - 1n)/2n;
91      }
92
93      public g(exp: bigint): bigint {
94          return this.exp(this.generator, exp);
95      }
96
97      public exp(base: bigint, exp: bigint): bigint;
98      public exp(base: Buffer, exp: Buffer): Buffer;
99      public exp(base: any, exp: any): any {
100         //DiffieHellman.computeSecret(base) raises "RangeError: Invalid key type" with DiffieHellman.setPrivateKey(0n.toBuffer())
101         if (exp === 0n) return 1n;
102         return super.exp(base.toBuffer(), exp.toBuffer()).toBigint();
103     }
104
105     public random(): bigint;
106     public random(): Buffer;
107     public random(): any {
108         //DiffieHellman.generateKeys() generates private keys of ${order}'s bit length rather than from the group
109         return super.random().toBigint() % this.order;
110     }
123     public inverse(x: any): bigint {
124         if (typeof x !== 'bigint') x = BigInt(x);
125         return (this.order - (x % this.order)) % this.order; //Equivalently, Fermat's (little) theorem could be applied.
126     }
127
128     public isMember(x: bigint): boolean {
129         //For a cyclic (sub)group of prime order, every group element (except identity) is a generator; raising a group generator (or identity) to the
130         //group's order produces one
131         //Here be dragons: DiffieHellman.computeSecret(base) raises "RangeError: Invalid key type" with DiffieHellman.setPrivateKey(this.order.toBuffer())
132         try { return this.exp(x, this.order-1n) * x % this.prime === 1n; } catch (error) { return false; }
133         //(Checking crypto.createDiffieHellman(this._prime, x.toBuffer()).verifyError !== 0 won't suffice; "don't worry if [x] is a generator or not:
134         //since we are using safe primes, it will generate either an order-q [subgroup] or an order-2q group [either] is OK" (OpenSSL dh_gen.c),
135         //see also https://florianjw.de/en/insecure_generators.html.)
136     }
137
138 }
```

Listing 7: Marshals Diffie-Hellman inputs/outputs between Buffer and bigint.

```
1  import crypto from 'crypto';
2
3  import { SigmaProtocol } from './ZKP';
4  import { SigmaProtocolDLog, Statement as _Statement, Witness, Commitment as _Commitment, Challenge, Response } from './ZKPDLog';
5
6  type Statement = _Statement & {a: bigint, d: bigint};
7
8  type Commitment = {G: _Commitment, A: bigint};
9
10 class SigmaProtocolDLogEq extends SigmaProtocolDLog implements SigmaProtocol<Statement,Witness,Commitment,Challenge,Response> {
11
12     public readonly statement: Statement;
13
14     constructor(statement: Statement, x?: Witness) {
15         super(statement as _Statement, x);
16         this.statement = statement;
17     }
18
19     public commit<T extends Commitment | _Commitment>(): T {
20         return {G: super.commit(), A: this.exp(this.statement.a, this.s) % this.prime} as T;
21     }
22
23     public verify(commitment: Commitment|_Commitment|any, challenge: Challenge, response: Response): boolean {
24         if (!commitment.G || !commitment.A) return false;
25         return super.verify(commitment.G, challenge, response)
26             && this.exp(this.statement.a, response) === commitment.A * this.exp(this.statement.d, challenge.toBigint()) % this.prime;
27     }
28
29 }
30
31 import { ZeroKnowledgeProof } from './ZKPDLog';
32
33 export class ZKPDLogEq extends ZeroKnowledgeProof {
34     constructor(statement: Statement, hashAlgorithm: string, witness?: Witness) {
35         super(new SigmaProtocolDLogEq(statement, witness), hashAlgorithm);
36     }
37 }
```

1. The prover chooses $s \in \mathbb{Z}_q$ at random and computes $(a, b) = (g^s, m^s)$. This pair is sent to the verifier.
2. The verifier chooses a random challenge $c \in \mathbb{Z}_q$ and sends it to the prover.
3. The prover sends back $r = s + cx$.
4. The verifier accepts the proof if

$$g^r = ah^c \quad \text{and} \quad m^r = bz^c.$$

Listing 8: Application of our framework deriving a zero-knowledge proof system by application of the Fiat-Shamir transformation to the sigma protocol by Chaum & Pedersen (overlay, Crypto'92), that sigma protocol being implemented by extension of the protocol by Chaum et al. (Listing 3), specifically, we extend types for a prover's statement and their commitment (Lines 6 & 8), generalise our method for computing a commitment to also include $a^s \bmod p$ (Lines 19–21), and extend verification (Lines 23–27) to hold iff we additional have $d \equiv a^x$, i.e., $\log_g h = \log_a d$. Deriving the zero-knowledge proof system is extremely easy (Lines 33–37). Such proofs may be used to prove correctness of partial decryptions: Suppose $(a, b) = (g^r, h^r \cdot m)$ is an ElGamal ciphertext such that $\log_g h = \log_a d$, we have $b \cdot d^{-1} \equiv g^{x \cdot r} \cdot m \cdot g^{-x \cdot r}$ where $x = \log_g h$, i.e., d is a partial decryption.

```
1  import crypto from 'crypto';
2
3  import { Group } from './Group';
4  import { SigmaProtocol } from './ZKP';
5
6  type Statement = {g: bigint, p: bigint, h: bigint, a: bigint, b: bigint};
7  type Witness = {r: bigint, m: 0|1};
8
9  type Commitment = { G: bigint; H: bigint }
10 type Challenge = Buffer;
11 type Response = {c: bigint; s: bigint};
12
13 export class SigmaProtocolDLogEqDisj extends Group implements SigmaProtocol<Statement,Witness,Commitment[],Challenge,Response[]> {
14
15     public readonly statement: Statement;
16     private readonly witness!: Witness;
17
18     private resp!: Response[];
19     private s!: bigint;
20
21     constructor(statement: Statement, witness?: Witness) {
22         super(statement.p, statement.g);
23         this.statement = statement;
24         if (witness) this.witness = witness;
25     }
26
27     public commit(): Commitment[] {
28         if (!this.witness) throw new Error("A witness must be instantiated to compute a proof");
29
30         const commitment = new Array<Commitment>(2);
31
32         //commitment for m
33         this.s = this.random();
34         commitment[this.witness.m] = {G: this.g(this.s), H: this.exp(this.statement.h, this.s)};
35
36         //simulate proof (commitment & response) for bit m xor 1
37         const c = this.random();
38         const t = this.random();
39         const G = this.g(t) * this.exp(this.statement.a, this.inverse(c)) % this.prime;
40         const H = this.exp(this.statement.h, t)
41                         * this.exp(this.statement.b * this.g(this.inverse(1^this.witness.m)) % this.prime, this.inverse(c)) % this.prime;
42
43         this.resp = new Array<Response>(2);
44         this.resp[1^this.witness.m] = {c, s: t};
45         commitment[1^this.witness.m] = {G, H};
46
47         return commitment as Commitment[];
48     }
```

1. The prover sets $a_v = g^w$ and $b_v = G^w$ for random $w \in_R \mathbb{Z}_q$. The prover also sets $a_{1-v} = g^{r_{1-v}} C_0^{d_{1-v}}$ and $b_{1-v} = G^{r_{1-v}} (U/G^{1-v})^{d_{1-v}}$, for random $r_{1-v}, d_{1-v} \in_R \mathbb{Z}_q$. The prover sends a_0, b_0, a_1, b_1 in this order to the verifier.
2. The verifier sends a random challenge $c \in_R \mathbb{Z}_q$ to the prover.
3. The prover sets $d_v = c - d_{1-v} \pmod q$ and $r_v = w - s d_v \pmod q$, and sends d_0, r_0, d_1, r_1 in this order to the verifier.
4. The verifier checks that $c = d_0 + d_1 \pmod q$ and that $a_0 = g^{r_0} C_0^{d_0}$, $b_0 = G^{r_0} U^{d_0}$, $a_1 = g^{r_1} C_0^{d_1}$, and $b_1 = G^{r_1} (U/G)^{d_1}$.

Listing 9: Applying our framework to the sigma protocol by Schoenmakers (overlay, Crypto'99), including a method (Lines 27–48) for computing a commitment comprising a pair, wherein element m defines parameters $g^s \pmod p$ and $h^s \pmod p$, and the other $g^t \cdot a^{-c} \pmod p$ and $h^t \cdot (b \cdot g^{-(m \text{ xor } 1)})^{-c} \pmod p$, where c, s, and t are randomly sampled. (Continues overleaf.)

```
50      public response(challenge: Challenge): Response[] {
51          if (!this.resp) throw new Error("Method response must be called after method commit");
52
53          this.resp[this.witness.m] = {} as Response;
54
55          //Here be dragons: Hash function produces far fewer bits than random number generation (for sensible length primes),
56          //apply function mod (rather than remainder operator) to negative bigint
57          this.resp[this.witness.m].c = (challenge.toBigint() - this.resp[1^this.witness.m].c).mod(this.order);
58          this.resp[this.witness.m].s = (this.s + this.witness.r * this.resp[this.witness.m].c) % this.order;
59
60          return this.resp as Response[];
61      }
62
63      public verify(commitment: Commitment[], challenge: Challenge, response: Response[]): boolean {
64          if (!this.isMember(this.statement.a) || !this.isMember(this.statement.b)) return false;
65
66          const proof = [0,1].map(i => ({G: commitment[i].G, H: commitment[i].H, c: response[i].c, s: response[i].s})).entries();
67
68          for (const [i, {G,H,c,s}] of proof)
69              if (!this.isMember(G) || !this.isMember(H)) return false;
70              else if ((this.g(s) % this.prime  !==  G * this.exp(this.statement.a,c) % this.prime)
71                  || (this.exp(this.statement.h, s) % this.prime
72                          !==  H * this.exp(this.statement.b * this.g(this.inverse(i)) % this.prime, c) % this.prime))
73                      return false;
74
75          if (challenge.toBigint() % this.order !== response.reduce((sum, {c}) => sum + c, 0n) % this.order) return false;
76
77          return true;
78      }
79
80  }
81
82  import { ZeroKnowledgeProof } from './ZKPDLog';
83
84  export class ZKPDLogEqDisj extends ZeroKnowledgeProof {
85      constructor(statement: Statement, hashAlgorithm: string, witness?: Witness) {
86          super(new SigmaProtocolDLogEqDisj(statement, witness), hashAlgorithm);
87      }
88  }
```

Listing 10: (Continued from Listing 9.) A method (Lines 50–61) returning a response comprising a pair, wherein element m `xor` 1 comprises values c and t computed by the commitment method and the other defines parameter c by subtracting the former's complementary value from the challenge and the remaining parameter as $s+r*c$. Verification (Lines 63–78) succeeds iff for each commitment pair G & H and response c & s we have $g^s \equiv G \cdot a^c \pmod{p}$ and $h^s \equiv H \cdot (b \cdot g^{-i})^c \pmod{p}$ where i is an index, and the challenge hash is the sum of c-values. Once again, deriving the zero-knowledge proof system is speculatory easy (Lines 84–88). Such proofs may be used to prove correctness of boolean encryption: Suppose $(a, b) = (g^r, h^r \cdot g^m)$ is an ElGamal ciphertext such that either $\log_g a = \log_h b$ or $\log_g a = \log_h b \cdot g^{-1}$, we have $m = 0$ in the former and $m = 1$ in the latter, i.e., m is a boolean. Chang-Fong & Essex (ACSAC'16) discovered the original presentation (Crypto'99) omits some necessary checks, we consider the patched version which additionally checks the statement and each commitment pair comprise group elements (Line 64 & 69).