1 HB and HB+

HB is an authentication protocols first introduced by Blum et al. [HB01], [BFKL94] that relies on the hardness of the learning parity with noise problem (LPN) for security and is provably secure against passive attacks.

A modification of the HB protocol in order for it to be secure against an active adversary is the HB+ protocol by Juels and Weis [JW05].

1.1 Protocols

Figure 1 shows one iteration of the authentication of HB.

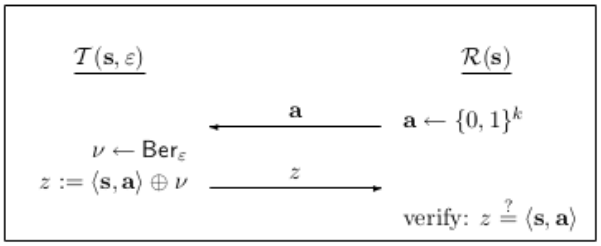


Figure 1: One iteration of the HB protocol

A tag $\mathcal{T}$ and a reader $\mathcal{R}$ share a random secret key $\boldsymbol{s} \in \{0,1\}^k$. One of the n iteration (all of which happen in parallel) of the authentication step consists of the following: $\mathcal{R}$ sends a random challenge $\boldsymbol{a} \in \{0,1\}^k$ to $\mathcal{T}$ who in turn calculates $\boldsymbol{z} := \langle \boldsymbol{s}, \boldsymbol{a} \rangle \oplus v$ with $v \leftarrow \mathsf{Ber}_\varepsilon$. This result is sent back to $\mathcal{R}$ who then calculates if the iteration is *successful*, i.e. $\boldsymbol{z} = \langle \boldsymbol{s}, \boldsymbol{a} \rangle$. Notice that even iterations of an honest tag using the correct key s can be unsuccessful with probability ε. The reader therefore accepts the authentication of the tag if the number of unsuccessful iterations is at most $\approx \varepsilon \cdot n$.

Figure 2 shows one iteration of the authentication of HB+.

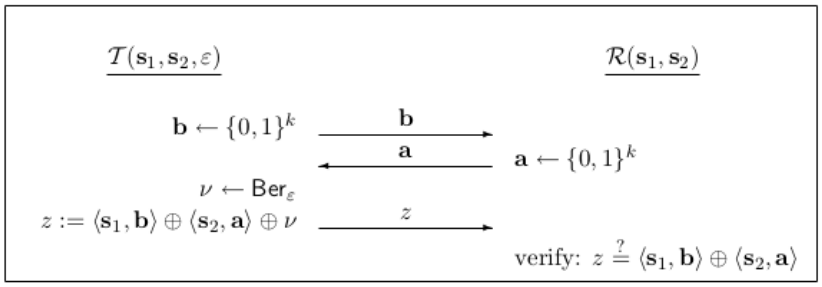


Figure 2: One iteration of the HB+ protocol

$\mathcal{R}$ and $\mathcal{T}$ now share two secret keys $\boldsymbol{s_1}, \boldsymbol{s_2} \in \{0,1\}^k$. One iteration of the authentication step now looks as follows: $\mathcal{T}$ first sends a random "blinding factor" $\boldsymbol{b} \in \{0,1\}^k$ to $\mathcal{R}$. The reader then, as for HB, sends a random challenge $\boldsymbol{a} \in \{0,1\}^k$ to $\mathcal{T}$ who in turn calculates $z := \langle \boldsymbol{s_1}, \boldsymbol{b} \rangle \oplus \langle \boldsymbol{s_2}, \boldsymbol{a} \rangle \oplus v$ with $v \leftarrow \mathsf{Ber}_\varepsilon$. This result is sent back to $\mathcal{R}$ who then calculates if the iteration is *successful*, i.e. $z = \langle \boldsymbol{s_1}, \boldsymbol{b} \rangle \oplus \langle \boldsymbol{s_2}, \boldsymbol{a} \rangle$. Again, even if $\mathcal{T}$ sends an honest

z using the correct keys $\boldsymbol{s_1}$, s_2 the iteration can be *unsuccessful*. Therefore, up to $\approx e \cdot n$ *unsuccessful* iterations are allowed fot the tag to still be accepted.

1.2 Implementation

We implemented different versions of HB and HB+ in order to investigate different degrees of parallelism and their effect on the performance on the Arduino Uno. For the parameters we worked with the suggested values given in [AHM14] and looked at 80-bit and 128-bit security.

First, we implemented the naive version where the Tag receives one challenge at a time to use with the first two sets of parameters (for HB and HB+ respectively). Then we implemented the improved version suggested in [GRS08] that required a slight modification to precompute r noise bits where the number of 1s is less than $u \cdot r$ to use the third set of parameters reducing the number of rounds to 256 [GRS08].

```
/*
 * Simulation of HB protocol using either a random key or the true key with
 * probability of 0.5 each.
 */
void hbTest()
{
  int counter = 0; // counter for unsuccessful iteration

  /* TAG: generate candidate key */
  uint8_t candidate[keySize];
  generateKey(&candidate[0]); //prints TRUE KEY or RANDOM KEY

  /* n iterations of HB */
  for(int i=0; i<n; i++) {

    uint8_t a[keySize]; // random challenge a in {0,1}^k

    boolean z; // response z in {0, 1}

    /* READER: choose random challenge a */
    for(int j=0; j<keySize; j++) {
      a[j] = random(256);
    }

    /* TAG: get z as candidate*a XOR v */
    z = getZ(candidate, a, keySize);

    /* READER: check if z = candidate* XOR v ?= key*a */
    if(z != dotProduct(key, a, keySize)) {
      counter++;
    }

  }
```

Figure 3: Code snippet from our implementation of HB. Bitstrings are stored in 8-bit unsigned integer arrays. A message about the result of the authentication is printed depending on the counter for unsuccessful iterations.

```
/*
 * Simulation of HB+ protocol using either a random key or the true key with
 * probability of 0.5 each.
 */
void hbpTest()
{
  int counter=0; // counter for unsuccessful iteration

  /* TAG: generate candidate key */
  uint8_t candidate1[keySize1], candidate2[keySize2];
  generateKey1(&candidate1[0]); //prints TRUE KEY or RANDOM KEY
  generateKey2(&candidate2[0]);

  /* n iterations of HB */
  for(int i=0; i<n; i++) {

    uint8_t b[keySize1]; // random blinding factor b in {0,1}^k

    uint8_t a[keySize2]; // random challenge a in {0,1}^k

    boolean z; // response z in {0, 1}

    /* TAG: choose random blinding factor b */
    for(int i=0; i<keySize1; i++) {
      b[i] = random(256);
    }

    /* READER: choose random challenge a */
    for(int i=0; i<keySize2; i++) {
      a[i] = random(256);
    }

    /* TAG: get z as candidate1*b XOR candidate2*a XOR v */
    z = dotProduct(candidate1, b, keySize1)^dotProduct(candidate2, a, keySize2)^generateNoiseBit();

    /* READER: check if z = candidate1*b XOR candidate2*a XOR v ?= key1*b XOR key2*a */
    if(z != dotProduct(key1, b, keySize1)^dotProduct(key2, a, keySize2)) {
      counter++;
    }

  }
```

Figure 4: Same for HB+.

Our parallel implementation of HB and HB+ introduces a parameter *maxChallenges* to set the number of challenges sent at a time. Notice that sending all challenges at once would be infeasible since we would have to send $k \cdot r$ which, in the case of $k = 512$ and $r = 441$ bits, would be $512 \cdot 441 = 225.792$ bits or 27 kB while the Arduino Uno only has 2 kB of SRAM. The highest number for *maxChallenges* would be 32 but since other variables of the program also occupy space, the realistic number can be found at around 22 for HB and 18 for HB+ assuming 80-bit security.

```
void hbTest()
{
  int counter = 0; // counter for unsuccessful iteration

  /* READER: generate candidate key */
  uint8_t candidate[keySize];
  generateKey(&candidate[0]); // prints TRUE KEY or RANDOM KEY

  // Begin
  for(int toSend = n; toSend > 0; toSend-=maxChallenges) {

    size_t bytesToSend, bitsToSend;

    if(toSend > maxChallenges) {
      bitsToSend = maxChallenges;
      bytesToSend = maxChallenges/8 + (maxChallenges%8 != 0);
    }
    else {
      bitsToSend = toSend;
      bytesToSend = toSend/8 + (toSend%8 != 0);
    }

    uint8_t r[bitsToSend][keySize]; // random challenges R

    /* READER: choose random challenges R */
    for(int i=0; i<bitsToSend; i++) {
      for(int j=0; j<keySize; j++) {
        r[i][j] = random(256);
      }
    }

    /* TAG: Calculate response vector z */
    uint8_t z[bytesToSend] = {0};
    matrixVectorProduct(&z[0], r, candidate, bitsToSend);

    for(int i=0; i<bytesToSend; i++) {
      for(int j=0; j<8; j++) {
        setBit(&z[i], j, getBit(z[i], j)^generateNoiseBit());
      }
    }
```

Figure 5: Parallel version of HB (there is also a similar implementation for HB+). Packages of a maximum of $maxChallenges = 22$ challenges are sent at a time.

2 AUTH

The AUTH protocol shown in Figure 6 was introduced by Kiltz et al. [KPV+17] and represents a two-round authentication protocol secure against active attacks and man-in-the-middle attacks, even in a quantum setting. The security of this protocol relies on the hardness of the *subspace LPN problem* which is reducible to the generic LPN problem. The authors also derived two message authentication codes (MACs) from LPN

2.1 Protocols

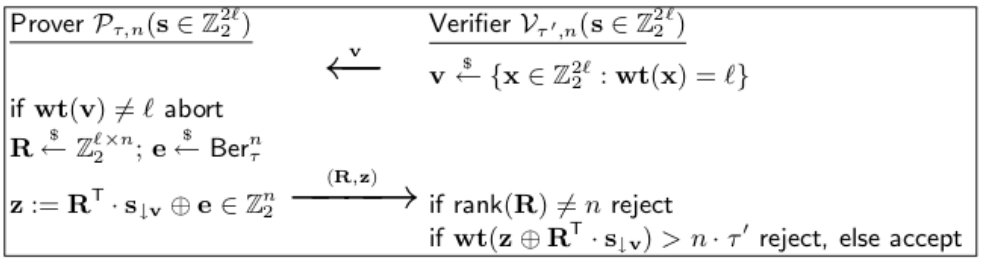


Figure 6: Two-round authentication protocol AUTH

Whereas the random challenges $\boldsymbol{R} \in \mathbb{Z}_2^{l \times n}$ (each row of the matrix $\boldsymbol{R}^T$ corresponding to one challenge a in HB) were computed by the Verifier $\mathcal{V}$ in HB, they are now computed by the Prover $\mathcal{P}$. $\mathcal{V}$ instead sends a random vector $\boldsymbol{v} \in \mathbb{Z}_2^{2l}$ with Hamming weight $\mathrm{wt}(\boldsymbol{v}) = l$ to select l of the $2l$ key bits of s to produce a key subset $\boldsymbol{s}_{\downarrow v}$ which is derived from $\boldsymbol{s}$ by deleting all bits $\boldsymbol{s}[i]$ where $\boldsymbol{v}[i] = 0$. Then, $\boldsymbol{z} \in \mathbb{Z}_2^n$ is computed as $\boldsymbol{R}^T \cdot \boldsymbol{s}_{\downarrow v} \oplus \boldsymbol{e}$ and sent to $\mathcal{V}$ along with $\boldsymbol{R}$. $\mathcal{V}$ rejects the authentication if either $\mathrm{rank}(\boldsymbol{R}) \neq n$ or if the number of unsuccessful iterations denoted as $\mathrm{wt}(\boldsymbol{z} \oplus \boldsymbol{R}^T \cdot \boldsymbol{s}_{\downarrow v})$ is greater than the threshold $n \cdot \tau'$ with $\tau' = 0.25 + \tau/2$.

2.2 Implementation

We implemented two versions of AUTH: the standard version given in Figure 6 and a variation that uses blinding vectors to avoid checking rank as given in [KPV+17]. Using a typical set of parameters $l = 500$, $n = 250$, $\lambda = 80$ we quickly realized that sampling a challenge matrix in $\mathbb{Z}_2^{l \times n}$ would be infeasible since one would need to store $500 \cdot 250 = 125.000$ bits or 15 kB while the Arduino only has 2kB of SRAM. We therefore deemed the protocol infeasible for our purpose of implementing it on the limited Arduino Uno together with MAC1 and MAC2 which use the same structure.

... Until we realized that we could split up the challenge matrix into chunks just like for HB/ HB+! We therefore implemented a version again using the *maxChallenges* parameter to specify how many challenges to send at a time. Through trial and error we arrived at a maximum of 15 challenges that we can send at the same time which yielded tagging times of ≈ 700 ms.

```
int checksum = 0;

for(int toSend = n; toSend > 0; toSend-=maxChallenges) {

  //Serial.println(toSend);

  size_t bytesToSend, bitsToSend;

  if(toSend > maxChallenges) {
    bitsToSend = maxChallenges;
    bytesToSend = maxChallenges/8 + (maxChallenges%8 != 0);
  }
  else {
    bitsToSend = toSend;
    bytesToSend = toSend/8 + (toSend%8 != 0);
  }

  /* __________ TAG __________ */

  // Compute R (R^T in the original version)
  uint8_t r[bitsToSend][keySize/2];

  for(int i=0; i<bitsToSend; i++) {
    for(int j=0; j<keySize/2; j++) {
      r[i][j] = random(256);
    }
  }
```

```
// Compute e as Ber_eps^m
uint8_t e[bytesToSend];

for(int i=0; i<bytesToSend; i++) {
  for(int j=0; j<(bitsToSend>=8 ? 8 : bitsToSend); j++) {
    boolean v = generateNoiseBit();
    setBit(&e[i], j, v);
  }
}

// Compute candidate_v
uint8_t candidate_v[C];
delete_v(candidate_v, candidate, v);

// Compute z = R^T * candidate_v XOR e in {0,1}^m
uint8_t z[bytesToSend] = {0};

matrixVectorProduct(z, r, candidate_v);

for(int i=0; i<bytesToSend; i++) {
  for(int j=0; j<(bitsToSend>=8 ? 8 : bitsToSend); j++) {
    setBit(&z[i], j, getBit(z[i], j) ^ getBit(e[i], j));
  }
}
```

Figure 7: Code snippet from our implementation of AUTH. Checksum is incremented for every failed challenge. The Code shows one iteration of sending m = *maxChallenge* challenges. R and e are sampled then z is calculated.

3 Lapin

3.1 Protocol

The protocol is defined over the ring $R = F_2[X]/(f)$. Both the honest tag $\mathcal{T}$ and the reader $\mathcal{TR}$ hold $s, s' \in R$ as the secret key, τ and τ' denote the bias for the Bernoulli distribution Ber_τ and the acceptance threshold. The two-round procedure is shown in 8.

$\mathcal{R}$ first sends a bitstring c of length λ to $\mathcal{T}$. $\mathcal{T}$ then samples an element r uniformly from the ring R and an element e that has its coefficients drawn independently from Ber_τ. $\mathcal{T}$ then calculates z as in Figure 8 where a mapping function $\pi : \{0,1\}^\lambda \to R$ maps c to an element of the ring. $\mathcal{R}$ finally receives r and z from $\mathcal{T}$ and counts the number of bits that are different between z and its own calculation of z without the noise. If the number is greater than $n \cdot \tau$ (n denotes the degree of f) then the authentication is rejected, otherwise it is accepted.

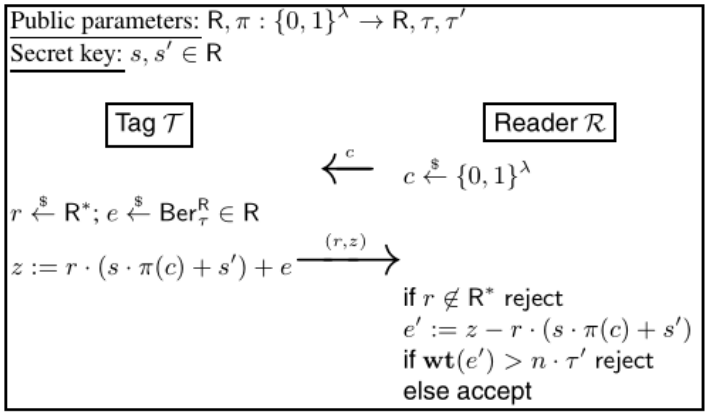
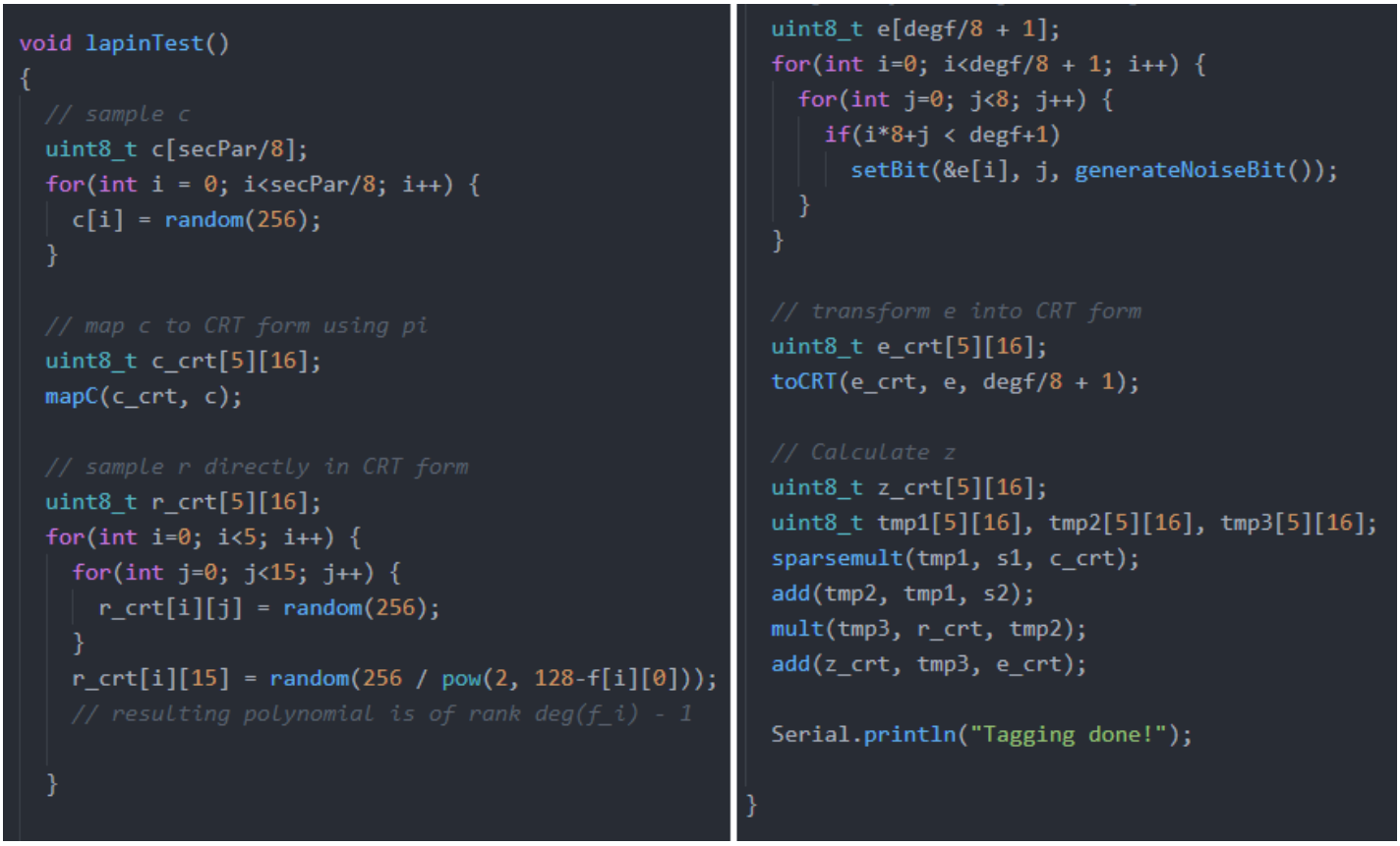


Figure 8: Two-round authentication protocol Lapin

3.2 Implementation

The original paper [HKL$^+$12] gives two possible implementations, one that uses a reducible polynomial for f and one that uses an irreducible one. We chose the former version because it is more efficient. Even though it requires a higher code size as a trade-off for faster response time, flash memory is not a scarce resource on the Arduino Uno with regards to this lightweight protocol.

Given that f factors into distinct irreducible factors $f_1, \ldots, f_m$, we can perform more efficient multiplications in the ring using the CRT (Chinese Remainder Theorem) representation $\hat{a} = (a \mod f_1, \ldots, f_m)$ of the elements in $F_2[X]/(f)$ which is why this version has an edge over the one using an irreducible f regarding performance. The paper suggests using a reducible polynomial f to define the ring $R = F_2[X]/(f)$ that is the product of $m = 5$ irreducible pentanomials where the numbers specify the coefficients that are non-zero: (127,8,7,3,0), (126,9,6,5,0), (125,9,7,4,0), (122,7,4,3,0), (121,8,5,1,0). The mapping $\pi : \{0,1\}^{80} \to R$ is defined as $\pi(c) = (v_1, \ldots, v_5)$ where we get $v_i \in F_2[X]/(f_i)$, $1 \leq i \leq 5$ by padding $c \in \{0,1\}^{80}$ with $deg(f_i) - 80$ zeros. Similar to the implementation laid out in the original paper, we used the right-to-left comb multiplication from [HMV05]. We also wrote a normal multiplication function and another sparse multiplication function for multiplication with $\pi(c)$ which is twice as fast as the former by abusing the fact that only the first 80 bits can be non-zero. Our implementation reached tagging times of ≈ 630 ms.



```
void lapinTest()
{
  // sample c
  uint8_t c[secPar/8];
  for(int i = 0; i<secPar/8; i++) {
    c[i] = random(256);
  }

  // map c to CRT form using pi
  uint8_t c_crt[5][16];
  mapC(c_crt, c);

  // sample r directly in CRT form
  uint8_t r_crt[5][16];
  for(int i=0; i<5; i++) {
    for(int j=0; j<15; j++) {
      r_crt[i][j] = random(256);
    }
    r_crt[i][15] = random(256 / pow(2, 128-f[i][0]));
    // resulting polynomial is of rank deg(f_i) - 1

  }
  uint8_t e[degf/8 + 1];
  for(int i=0; i<degf/8 + 1; i++) {
    for(int j=0; j<8; j++) {
      if(i*8+j < degf+1)
        setBit(&e[i], j, generateNoiseBit());
    }
  }

  // transform e into CRT form
  uint8_t e_crt[5][16];
  toCRT(e_crt, e, degf/8 + 1);

  // Calculate z
  uint8_t z_crt[5][16];
  uint8_t tmp1[5][16], tmp2[5][16], tmp3[5][16];
  sparsemult(tmp1, s1, c_crt);
  add(tmp2, tmp1, s2);
  mult(tmp3, r_crt, tmp2);
  add(z_crt, tmp3, e_crt);

  Serial.println("Tagging done!");

}
```

Figure 9: Code snippet from our implementation of Lapin.

References

[AHM14] Frederik Armknecht, Matthias Hamann, and Vasily Mikhalev. Lightweight authentication protocols on ultra-constrained rfids - myths and facts. In Nitesh Saxena and Ahmad-Reza Sadeghi, editors, *Radio Frequency Identification: Security and Privacy Issues*, pages 1–18, Cham, 2014. Springer International Publishing.

[BFKL94] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 278–291, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[GRS08] Henri Gilbert, Matthew J. B. Robshaw, and Yannick Seurin. : Increasing the security and efficiency of. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 361–378, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[HB01] Nicholas J. Hopper and Manuel Blum. Secure human identification protocols. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 52–66, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[HKL+12] Stephan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. Lapin: An efficient authentication protocol based on Ring-LPN. In Anne Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012*, volume 7549 of *Lecture Notes in Computer Science*, pages 346–365, Washington DC, United States, March 2012. Springer.

[HMV05] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. Guide to elliptic curve cryptography. *Computing Reviews*, 46(1):13, 2005.

[JW05] Ari Juels and Stephen A Weis. Authenticating pervasive devices with human protocols. In *Annual international cryptology conference*, pages 293–308. Springer, 2005.

[KPV+17] Eike Kiltz, Krzysztof Pietrzak, Daniele Venturi, David Cash, and Abhishek Jain. Efficient authentication from hard learning problems. *Journal of Cryptology*, 30(4):1238–1275, Oct 2017.