Securing Code and Zero-Knowledge

Anonymous
Rochester Institute of Technology

ABSTRACT

Zero-knowledge protocols offer large advancements in the context of data privacy. This paper presents an implementation of a zero-knowledge protocol for the discrete logarithm. The implementation is analyzed to discuss relevant secure coding concerns in relation to zero-knowledge. This paper also discusses the merits and pitfalls of zero-knowledge protocols in relation to secure coding best practices. This paper also explores the legal and ethical ramifications of zero-knowledge protocols.

KEYWORDS

Zero-Knowledge Protocols, Secure Coding

ACM Reference format:
Anonymous. 2019. Securing Code and Zero-Knowledge. In *Proceedings of NA, NA, 2019,* 7 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW

The amount of data produced and consumed on a daily basis has grown exponentially in the past few years. Individual's posts to a wide range of social media sites, search queries to Google and other search engines, and accesses to services such as the Weather Channel forecast have all become ubiquitous in the modern day. One study estimated that in 2018, there was approximately 2.5 quintillion bytes of data created each day [18]. This study also estimated that approximately 90% of data ever tracked was produced in in the previous two years. With an ever-increasing rate of data production and consumption, many people have begun to worry about the privacy of their own data. These worries are not unfounded as stories such as "How Target Figured Out A Teen Girl Was Pregnant Before Her Father Did" become even more common [16].

While many services are becoming more invasive and pervasive in our lives, there has been a counter-push in the last few years to reimagine the way that software integrates into our daily lives in order to give people more control over their own data. In this paper, we discuss one such technique for increasing data privacy: zero-knowledge. We explore the history of zero-knowledge protocols in Section 2. In Section 3 we provide an example of a zero-knowledge protocol. We explain the details of implementing a zero-knowledge proof for the discrete logarithm in Section 4. In Section 6 we discuss benefits and drawbacks of zero-knowledge algorithms with respect to secure coding. We explore the legal and ethical concerns related to zero-knowledge in Section 7. We describe possible further considerations in Section 8.

NA, NA
2019. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

Zero-knowledge proofs were first presented in the seminal paper "The Knowledge Complexity of Interactive Proof Systems" by Goldwasser, Micali, and Rackoff back in 1989 [15]. Zero-knowledge proofs are a form of interactive proof. In the interactive proof model, one agent with unlimited computing power must convince another agent with high probability whether or not a given string in a given language. The resulting model is very similar to a probabilistic version of NP with the key addition that the two agents may interact and keep coin tosses secret. A much celebrated result in complexity theory proved that PSPACE = IP (the set of languages decidable with interactive proof systems) [12].

Zero-knowledge proofs are a specific form of interactive proof where the prover wishes to enforce that the only information the verifier recieves throughout the interactive proof is that the statement provided by the prover is true. The interactive proof functions in such a way that a verifier, knowing only publicly available information, is able to verify that the prover has access to some fixed private information.

The most common application of zero-knowledge proofs is for a prover to convince a verifier that they know a password without providing even a cryptographic hash of the password to the verifier. Recent applications of zero-knowledge protocols have extended to other fields. Notably, zero-knowledge proofs have been applied in conjunction with blockchain technology to create anonymous cryptocurrencies where transactions can be verified without disclosing the amount of money transferred [19, 22]. Protocols have been designed to use zero-knowledge proofs in political domains as well. Zero-knowledge proofs allow voting systems to circumvent administrator fraud in anonymous voting mechanisms [25]. There is even research into analog versions of zero-knowledge protocols for testing nuclear residue without disclosing protected information [17].

The first zero-knowledge proof was constructed to show a verifier that a prover knows the modular square roots of a set of public keys in polynomial time [15]. Since no polynomial time algorithm is known for computing modular square roots without having access to a prime factorization of the modulus, a verifier cannot learn the exact modular square roots. Instead the prover provides information that can only computed in polynomial time if the prover actually knows the modular square roots. Newer research has since shown that under the assumption of a secure encryption, every language in NP has a zero-knowledge proof [3]. Other research has addressed the question of concurrency in establishing proofs by weakening zero-knowledge to witness-indistinguishability where no verifier can distinguish between provers which use different witnesses to prove a statement [14]. Current research in zero-knowledge explores the questions of minimal proof size and minimal setup [5, 27].

While new zero-knowledge protocols are fairly intricate, researchers have created multiple black-box implementations of zero-knowledge protocols for use in industry [2, 4, 6, 7, 9, 26, 27]. Most of these implementations are very new and not yet received much interest from industry.

3 ZERO-KNOWLEDGE EXAMPLE

An interactive proof protocol is said to be a zero-knowledge proof if it satisfies the following three properties: completeness, soundness, and zero-knowledge. Completeness implies that when a statement is true, that the verifier will be convinced by proof that the statement is true. Soundness implies that a prover cannot convince a verifier of a false statement except with some very small probability. Zero-knowledge requires that the verifier cannot learn anything beyond whether a statement is true or false.

A very simple example of a zero-knowledge protocol was given by Kostas Konstantin in a LinkedIn post promoting the use of zkSNARK algorithms (a non-interactive form of zero-knowledge proof) [11]. The example works as follows. Suppose Alice wishes to prove to her colorblind friend, Bob, that she can distinguish a red ball from a green ball. Without ever stating which ball is which color, Alice can convince Bob that she can tell the difference between the two colors by telling Bob whether or not he swapped two balls behind his back. In this example, the protocol runs as follows:

(1) Bob shows Alice both balls with one in his left hand and one in his right hand.
(2) Bob then shuffles the two balls behind his back tracking whether he swapped the balls or not.
(3) Alice announces only whether or not Bob has swapped the balls.
(4) Bob and Alice repeat steps (2) and (3) until Bob is sufficiently convinced that Alice is not guessing.

This protocol is complete because if Alice can actually distinguish the color of the two balls then she can always correctly tell Bob whether or not he swapped the balls. The protocol is sound because if Alice cannot distinguish the two balls (e.g. both of the balls are green), then the best she can hope to do is to guess whether or not Bob swapped the balls. Supposing that Alice does not know whether Bob will swap the balls, she has a probability of correctly guessing whether or not Bob swapped the balls of $1 - 2^{-n}$ where steps (2) through (4) are run for n iterations. The protocol is zero-knowledge because Bob does not learn anything about how to distinguish the balls. Bob only learns that Alice can distinguish the two balls.

4 ZERO-KNOWLEDGE DISCRETE LOG

While zero-knowledge protocols have existed since 1989, they have only recent started to be applied in commercial applications. One major factor in their scarcity is the difficulty explaining the concepts in a zero-knowledge proof. As a case study, we implemented a simple zero-knowledge protocol for proving knowledge of a discrete logarithm which was first presented by Chaum et al. in "An improved protocol for demonstrating possession of discrete logarithms and some generalizations" [13]. This algorithm relies on the apparent computational hardness of computing a discrete logarithm. [1] The protocol for proving knowledge of a discrete logarithm runs as follows:

(1) Given a generator g and a prime number p, Alice computes the value $y = g^x \mod p$ and transfers y to Bob.
(2) Alice computes a random number r. She then computes $C = g^r \mod p$ and transfers C to Bob.
(3) Bob chooses either to ask Peggy for r or for $(x+r) \mod (p-1)$. Depending on which information is asked for, he then verifies Peggy's information as follows:
    (a) If Bob asks for r, he verifies that $C \equiv g^r \mod p$.
    (b) If Bob asks for $(x + r) \mod (p - 1)$, he verifies that $Cy \mod p \equiv g^{(x+r) \mod (p-1)} \mod p$.
(4) Alice and Bob repeat steps (2) and (3) until Bob is sufficiently convinced that Alice is not guessing.

The above algorithm is complete because if Alice knows the true value of x, then she can always answer Bob's request correctly. If Bob asks for r, then he may verify that $C = g^r \mod p$ efficiently. Likewise, if Bob asks for $(x + r) \mod (p - 1)$, Alice can compute this value. The proof that Bob can simply verify $Cy \mod p \equiv g^{(x+r) \mod (p-1)} \mod p$ is omitted from the Wikipedia explanation. This follows from the fact the following argument:

$$\begin{aligned} Cy \mod p &\equiv g^{x+r} \mod p \\ &\equiv g^{(x+r) \mod (p-1)} g^{k(p-1)} \mod p \\ &\equiv g^{(x+r) \mod (p-1)} 1^k \mod p \\ &\equiv g^{(x+r) \mod (p-1)} \mod p \end{aligned}$$

The middle step follows from Fermat's Little Theorem.

If Alice does not know the value of x, she can attempt to lie for either r or $(x + r) \mod (p - 1)$. If she lies for r, then she can correctly compute $C = g^r \mod p$ and transmit that information to Bob. However, when asked the value of $(x + r) \mod (p - 1)$, she will not be able to provide the correct value as she cannot compute the discrete logarithm of Cy for base g in the modulus p. If Alice tries to lie for $(x + r) \mod (p - 1)$ then she can provide a new random number r' and disclose $C' = g^{r'}(g^x)^{-1} \mod p$ to Bob. Bob will then compute $Cy = g^{r'}$ correctly. However, if Bob asks Alice for r, then Alice cannot provide the correct value of r for which $g^r \mod p = C'$ as this would require computing the discrete logarithm of C' base g in the modulus p. In either case, Alice has only a 50% probability of correctly guessing what Bob will ask her for before she has to provide some value for C. Thus is the algorithm is run for n iterations, Bob will know that Alice knows the value of x with certainty $1 - 2^{-n}$. Thus this protocol is sound.

This algorithm is a zero-knowledge protocol as Bob never learns the value of x given $g^x \mod p$. As Bob cannot efficiently compute the discrete logarithm of $g^x \mod p$, he only learns that Alice does in fact know the value of x.

Our implementation of a zero knowledge proofs for the discrete logarithm runs in Java, and it uses a peer-to-peer framework where a client wants to convince a server that they know some private

[1] The exact complexity of computing a discrete logarithm is still unknown. It is one of the candidate problems for NP-Intermediate, the class of problems which in NP but are neither in P nor in NP-Complete.

Figure 1: Generating a large prime number

```
long genProbPrime() {
    while(true) {
        long rounds = 128;
        int n = 16;
        long range = (long) Math.pow(2, n);
        long p = Math.abs(random.nextInt()) % range;
        p += range;
        if(p % 2 == 0){
            p += 1;
        }
        if (millerRabin(p, rounds))
            return p;
    }
}
```

Figure 2: Fast Modular Exponentiation

```
long modPow(long a, long d, long n) {
    if(d == 0) {
        return 1;
    }
    if(d % 2 == 0) {
        return modPow((a*a)%n, d/2, n);
    }
    return (a * modPow((a*a)%n, d/2, n)) % n;
}
```

Figure 3: Computing Necessary Iterations

```
int necessaryIterations(double min_threshold) {
    double max_fail = 1 - min_threshold;
    int iters = 1;
    double prob_fail = .5;
    while (prob_fail > max_fail) {
        iters++;
        prob_fail /= 2;
    }
    return iters;
}
```

password information. The implementation has been published to Github and is available to the public [23].

In order to implement a zero-knowledge protocol for the discrete logarithm, there are many non-trivial things that have to be considered. While the discrete logarithm proof functions for any prime modulus p, it is important that p be a large prime number as smaller numbers offer an easy attack channel using a brute force technique to compute a discrete logarithm. Furthermore, this is not the only requirement on p for securely implementing the discrete logarithm zero-knowledge protocol. One very standard approach for generating large prime numbers is to use primes of the form $2^n - 1$ or $2^{2^n} + 1$ (Mersenne primes and Fermat primes respectively). However, it has been shown that both Mersenne primes and Fermat primes give way to polynomial time algorithms for computing a discrete logarithm [20, 24]. In order for the zero-knowledge proof to be meaningful, p must be chosen to avoid these reductions in complexity. In our implementation, we used the algorithm in Figure 1 to generate a large prime number. The algorithm generates a random odd number in the range $[2^n, 2^{n+1})$. Since there are approximately $\frac{n}{\log n}$ prime numbers less than n, the probability that a random odd number in this range is prime is approximately $\frac{1}{n}$. This means that in order to find an n bit prime number, it is expected that one will be found within n the first n randomly generated odd numbers. It is important to note that the prime p generated by this procedure will always fit in a long data type (we fixed $n = 16$ for demonstration purposes). However, for more secure applications, the data type of p should support primes of larger length (e.g. 128 bits or more is usually recommended).

In order to test primality efficiently, the generating algorithm uses the Miller-Rabin primality test. Miller-Rabin functions will always return true if the input number is prime, but with high probability (after 128 rounds the probability is about $1 - 2^{-128}$) will return false if the input number is not prime. It is theoretically possible for the Miller-Rabin test to fail, but the probability of this occurring is negligible.

The next challenge in efficiently implementing a zero-knowledge protocol for the discrete logarithm is efficient modular exponentiation. The direct method of computing an exponent and then computing the mod will either suffer from overflow or use an unnecessary amount of space to store intermediate values. Instead, the modulus should be taken at each multiplication to prevent overflow errors. Another consideration is the time complexity inherent in computing x^y. Multiplying x a total of y times will take time linear in the value of y and exponential in the length of the input. To make this efficient, we instead use a repeated squaring technique. Using the binary representation of y, we can represent x^y as a series of operations of the form either $x * a^2$ or a^2. This algorithm is shown in Figure 2. We observe that in our implementation, we were careful to enforce that a and n are both less than 2^{32} in order to prevent overflows where a^2 or $a * n$ is larger than the maximum value of a `long` in Java. This notably limits the maximum value of p to 2^{32}, so for a more secure application it is important that the data types used for modular exponentiation also scale to support approximately p^2 without overflow.

The last non-trivial part of implementing the zero-knowledge protocol is to determine how many iterations are necessary to provide a threshold probability for the verifier. Suppose the verifier will accept a proof when there is at least probability q that the prover isn't cheating. Given q, it is necessary to find the smallest value of n such that $(1 - 2^{-n}) \geq q$. Our implementation uses the code shown in Figure 3. This code will correctly compute a minimum number of iterations for any valid threshold (greater than 0 and at least the floating point unit less than 1). However, from a code security standpoint, this range is never tested, and the code will run forever if `min_threshold` is greater than or equal to 1. An alternate approach would be to use the solution to the equation $1 - 2^{-n} \geq q$ which gives $n \geq \log_2(1 - q)$. In this case, error checking is still necessary as $\log_2(x)$ is undefined when $x \leq 0$.

5 EXPLOITING MATHEMATICAL INSECURITIES

In order to demonstrate the level of security provided by zero-knowledge, we implemented multiple attacks against our implementation of the zero-knowledge proof of the discrete logarithm. As illustrated above in Section 4, if a malicious prover knows what the verifier will ask for they may commit to a strategy accordingly. For a benchmark comparison, we implemented a prover which randomly guesses what the verifier will ask for. As expected, this strategy proved fairly inefficient at falsifying "proofs." All tests in our experiment assumed the verifier requires at least 95% confidence before accepting a proof (equivalently, all tests ran 5 iterations of the protocol). This means that in expectation, it will take approximately 32 iterations of randomly guessing a proof before the verifier will accept a false proof.

The first attack strategy that we implemented is for the prover to brute force the secret information that they are trying to convince the verifier that they have knowledge of. For the discrete logarithm (with prime modulus between 2^{16} and 2^{17}) this turns out to be very feasible. The brute force algorithm continually raises g to successive powers mod p until it eventually finds a value such that $g^x \mod p \equiv y$ for the value of y that was already committed. (In practice, y would be provided by the server to prevent the prover from generating a new secret key each time.[2]) The brute force algorithm takes time $O(p)$, but since p is a numeric input represented in binary, it takes time exponential in its input. Since p is small in our implementation, the brute force algorithm is sufficient for computing a discrete logarithm. In practice, p should be significantly larger so that this brute force algorithm will take much more time. As demonstrated above, finding an n bit prime number can be done in linear expected time.

To speed up the brute force attack, we also implemented a slightly more complicated algorithm for computing the discrete logarithm using a brute force idea. The algorithm we implemented is known as the Baby-step Giant-step algorithm. The idea behind the algorithm is that if $m = \lceil\sqrt{p}\rceil$, then it is possible to decompose x uniquely into $im + j$ where $0 \leq i < m$ and $0 \leq j < m$. By computing all values of g^j in time $O(m)$, it is possible to check if yg^{-im} is in the set of $\{g^j\}_{j=1}^m$ in constant time for each i. Thus in time $O(m) = O(\sqrt{p})$, it is possible to find which i (and thus which j) results in $g^{im+j} = y \mod p$. As our implementation used small prime numbers, this approach is feasible for within reasonable time frames. However, if the prime numbers were scaled up in magnitude, then a simple timeout would cause this approach to be infeasible as no polynomial time algorithm is known for computing the discrete logarithm.

The next attack scheme we implemented scales significantly better than the above attack, but it requires significantly more knowledge about the implementation of the protocol. By leveraging exposed calls to a pseudorandom number generator, it is possible to brute force compute the seed currently used by the random number generator. This in turn makes it possible to guess the subsequent values of random number generator. In the case of the zero knowledge proof, this ability to predict "random" numbers can give a cheating prover a distinct advantage over an honest verifier. As shown in the proof of the zero knowledge protocol, a prover may first guess what the verifier will ask for and commit to a strategy accordingly. If the guess is random, the prover will succeed with probability 2^{-n} over n rounds. However, suppose the prover can guess what the verifier will ask for with probability p. This yields a probability of terminating the proof with a false positive of p^{-n}. For any p larger than $\frac{1}{2}$, this might cause a verifier to accept a proof before they should. (A verifier with knowledge of p can request a number of rounds proportional to $-\ln tol/\ln p$ where tol is the tolerance required by the verifier). However, if a prover knows with probability 1 what the verifier will ask for, then a cheating prover may always provide a false positive proof.

Figure 4: Java's Random.next()

```
protected int next(int bits) {
    long oldseed, nextseed;
    nextseed = (this.seed * multiplier + addend) & mask;
    this.seed = nextseed;
    return (int)(nextseed >>> (48 - bits));
}
```

In order to crack Java's pseudorandom number generator, it is sufficient to know only two subsequent calls to `Random.nextInt()`. Since the first value exposes bits 16 to 48 of the seed used for the second call, one most only check 2^{16} possible combinations of bits 0 to 15 to determine which results in the correct value of `Random.nextInt()`. Java's pseudorandom generator uses a linear congruential generator, which means that it follows the update rule in Figure 4. The values of `mask`, `multiplier`, and `addend` are all constants which are easy to find.[3] While the linear congruential generator is not truly random, assuming it is simplifies the computation of uniqueness in the results of cracking the random number generator. Suppose that from the value of the pseudorandom seed all but x bits are exposed to a user and that after calling `Random.next()`, y output bits are exposed to the user in the result. If the user knows which x bits are unexposed, it is possible brute force check all combinations of the x bits to determine which input seeds result in the value containing the y exposed bits. One important question is how many such input seeds yield the resulting output. There are 2^x possible seeds which yield 2^y outputs. If the pseudorandom number generator acts uniform randomly, then the distribution of the inputs to the outputs will be uniform. Thus there will be $2^x/2^y = 2^{x-y}$ input seeds per output. For example, consider attacking `Random.nextInt()` using this technique. `Random.nextInt()` exposes 32 out of 48 bits. Thus $x = 16$ and $y = 32$. This means that in expectation, for a given output, there is a unique input seeds which yielded that output (the expected number of input seeds which yield any output seed is less than 1).

Observe that this same strategy is unlikely to succeed when predicting a sequence of boolean values. `Random.nextBoolean()` exposes 1 out of 48 bits. This means that a single pair of boolean values generated sequentially leads to approximately $2^{47-1} = 2^{46}$ expected

[2] In our implementation, the cheating prover commits a random value for $y \mod p$. Since the "generator" is chosen randomly rather than by picking a true generator for the group $\mathbb{Z}_p^*$, there is a chance that there does not exist a value x such that $g^x \equiv y \mod p$. Since $\mathbb{Z}_p^*$ contains $\phi(p-1)$ generators where ϕ is Euler's totient function, the probability that a random generator is truly a generator is $\phi(p-1)/p$.

[3] `mask` = 2^{48}, `addend` = 11, and `multiplier` = 0x5DEECE66DL

candidate seeds for the initial call to `Random.nextBoolean()`. However, note that progressive calls to `Random.nextBoolean()` are not independent. Thus each successive call exposes 1 more bit of information. This means that after approximately 47 calls to `Random.nextBoolean()`, the input seed which generates the observed sequence of 47 boolean values will likely be unique. A cheating prover may thus treat each successive guess as truly random until they have cracked the sequence of booleans provided to them. This will occur after 47 iterations; however, by this time they already will have a probability of success less than 2^{-47}. If the verifier does not ask for at least 47 rounds, then the cheating prover gains no value from this technique.

While it may seem fruitless, this strategy is not entirely flawed. First, the pseudorandom number generator is not truly random. Thus by tracking the candidate seeds that survive each successive round of bit exposure, it might be possible to predict the next output with probability slightly larger than $\frac{1}{2}$. Consider a policy which either guesses by majority vote or by random vote according to the next boolean that would be observed by each seed in the candidate pool. If it is not the case that exactly half of the seeds vote true and half vote false, then the probability of error on each guess can be reduced to the probability of the majority vote being correct. Conditioned on the correct seed being in the majority, this value is more than $\frac{1}{2}$ at each round. While this strategy does not guarantee a reduction in error rates, it does often reduce error in practice.

Beyond tracking candidate guesses, there is more information exposed in order to run the discrete logarithm zero-knowledge protocol. In particular, our implementation relies upon a random generator and large prime number. The computation of a prime number makes a sequence of calls to `Random.next()` before exposing an undetermined number of bits to the user in the form of the output prime. However, if the pseudorandom number generator's seed that was used to produce the prime number is known, then the process of computing the random prime number is deterministic and repeatable. This leads to a much stronger attack vector where the exposed random generator value and the exposed prime number offer sufficient information to reduce the probability of failure to a much more acceptable attack. We first observe that the call to `Random.nextInt()` and the subsequent absolute value call for computing a generator lead to an exposure of all but 17 bits of information (the sign bit is lost by the absolute value, so it must also be recovered in the brute force search). In order to determine how much information is exposed by the call to `ZKMath.genProbPrime()` in our implementation of the zero-knowledge protocol, we evaluate the ratio of the number of inputs to the number of outputs. Assuming that the random inputs yield a uniform distribution over the the outputs gives an expected number of candidate seeds for each possible observed pair of input generator and prime number. Since `ZKMath.genProbPrime()` generates a prime number between 16 and 17 bits long, there are $\pi(2^{17}) - \pi(2^{16}) = \Theta(\frac{2^{17}}{17} - \frac{2^{16}}{16}) = \Theta(2^{12})$ prime numbers in this range where π is the prime-counting function.[4] Given 2^{17} possible candidate seeds which map to 2^{12} output primes, there are approximately $2^5 = 32$ candidate seeds per sampled prime given a fixed generator. This means that after approximately 5 rounds, a cheating prover could guarantee any subsequent rounds are answered correctly. For the case that a verifier only requires 5 rounds, this strategy does not provide significant benefit to the cheating prover, but it provides significantly better expected results for some modifications to the protocol. For example, if the length of the prime number increases (i.e. if instead of using 16 bit primes, 20 bit primes are used), the number of candidate seeds per prime actually decreases in expectation. This means that the as the protocol becomes more secure to a brute force attack in computing the discrete logarithm, the likelihood of finding a unique seed for random (from a fixed number of possible seeds) actually increases too. This implies that as the size of the prime number used increases, the less secure the implementation is to this type of attack. Also observe that for only 5 iterations, this random number cracking approach barely outperforms randomly guessing. However, by removing independence from future iterations, this strategy will stay at 2^{-5} probability of success while random guessing will continue to approach 0 with a probability of success of 2^{-n}.

For a comparison of each of the approaches, we ran 1000 iterations of the zero-knowledge protocol for 10 iterations. The number of proof attempts which survived each round was recorded for each approach. We also recorded the total time of all 1000 proofs. For comparison, the expected number of proofs which survive each round is given as follows:

- Random Guessing: $1000(2^{-n})$ survive each round. This is less than 1 for all 10 rounds.
- Discrete Logarithm Cracking: if g is a generator of the group $\mathbb{Z}_p^*$ then this will always succeed. If g is not a generator, then this acts as well as guessing that the verifier will ask for r at each round (observe that this is equivalent to randomly guessing in expectation).
- Cracking the random sequence: if s is the number of unexposed bits in generating g and t is logbase 2 of the number of possible generated primes, then when $n < s-t$ there will be $1000(2^{-n})$ surviving proofs. For $n \geq s - t$, this value is expected to stay constant with $1000(2^{s-t})$ surviving proofs. For our implementation $s - t \approx 5$.

For the average time per proof, there is a balance between the number of rounds that the proof survived and the amount of overhead before the proof is run. It is perhaps interesting to observe that despite the additional overhead involved in running the Baby Giant algorithm, the increased rate of failure (and thus skipped rounds) led to a decreased average time overall. The results of our experiments are shown in Figure 5 and 6 respectively.

In terms of feasibility in practical settings, a brute force approach would likely be too slow to run as most real world cryptographic protocols use significantly larger values than our example. In contrast, cracking the random sequence of guesses would also be infeasible as it largely requires knowledge of the calls to random that a target system has made. However, it is important to note that security through obscurity violates the principle of open design in secure coding, so a truly secure system would need a mechanism in place to make sure that this approach isn't feasible even with visible calls to random. For example, adding spurious calls to random or

[4] In general for prime numbers between 2^n and 2^{n+1}, there are about $\pi(2^{n+1}) - \pi(2^n) = \Theta(\frac{2^{n+1}}{n+1} - \frac{2^n}{n}) = \Theta(\frac{2^n}{n}) = \Theta(\pi(2^n)) \approx 2^{n-\log_2(n)}$ primes in this range.

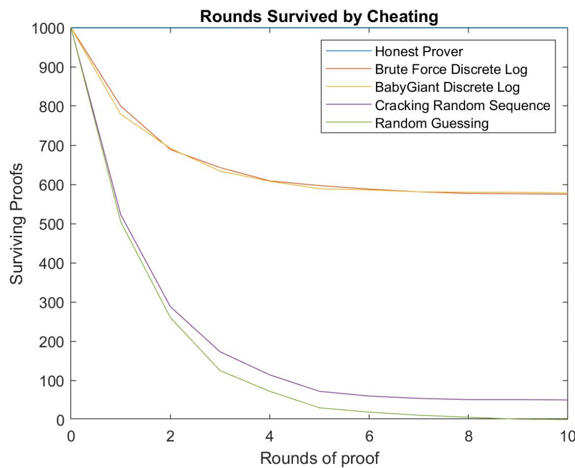

Figure 5: The number of surviving proofs after successive iterations under different strategies.

Figure 6: The average time per proof of different strategies

Strategy	Average Time per Proof (ms)
Honest Proover	0.745
Brute Force Discrete Log	1.691
BabyGiant Discrete Log	0.704
Cracking Random Sequences	6305
Random Guessing	0.302

resetting the random seed using the system time every so often both would mitigate the risk of an attack via random cracking.

6 SECURE CODING IMPLICATIONS OF ZERO-KNOWLEDGE

The primary goal of zero-knowledge is to reduce the amount of data exposed while transferring knowledge. In a classical sense, it previously would have been necessary to disclose one's age in order for a verifier to know that they fall within the ages of 18-65. However, in this example, if there is no need for the prover to disclose their age, then doing so violates of the principle of least privilege as proposed by Saltzer and Schroeder [21].

The existence of zero-knowledge proofs forced security experts to reimagine what constitutes a truly minimal set of necessary privileges. Zero-knowledge range queries and set membership queries reshape the classical conventions minimal data [8, 10]. For example, it is now possible to prove that someone is between 18 and 65 years old without disclosing their actual age. You can prove that someone is within Europe without disclosing their exact location [8]. Even more practically, one can prove validity of banking transactions without knowing the actual amounts of money within the relevant accounts [19, 22].

While least privilege is central to the design of zero-knowledge protocols, other secure coding principles are in play when considering zero-knowledge. At its current state, zero-knowledge protocols offer huge potential to support open design. The basic protocols are designed so that no player, even knowing how the protocol functions, can cheat the system without an accomplice. In the colored balls example in Section 3, it would be possible for Alice and Bob to convince another colorblind agent that they can distinguish two green balls by agreeing on the sequence of swaps ahead of time. Newer research has removed this failure point by introducing non-interactive zero-knowledge proofs [11]. The other major failure point in the open design of zero-knowledge proofs is the lack of concurrency support. With a series of coordinated verifiers, the simple zero-knowledge protocols can leak information to the verifiers. This too has recently been addressed by weakening zero-knowledge to witness-indistuingishability [14].

As black-box implementations of zero-knowledge protocols are becoming more prevalent and more powerful, the psychological acceptability of zero-knowledge proofs is improving as well. Since zero-knowledge protocols don't require any additional knowledge of the user, a proper implementation will make resources harder to access than not using any security mechanism. Even for cases like password authentication, a zero-knowledge proof of knowledge is no more complicated for a user than a standard password authentication protocol.

Zero-knowledge proofs, however, do have some issues with respect to code security. While the probability of failure can be made arbitrarily low using repeated iterations, it is never possible to guarantee 100% failure for when a prover cheats. This concern represents negligible risk as long the probability of failure is negligible. Furthermore, as practical and applied zero-knowledge techniques are still relatively new, the current designs are somewhat complicated to explain. This complexity represents a failure of economy of mechanism where there are simpler alternatives to zero-knowledge for accomplishing some tasks. In comparison to the benefits of least privilege, this increased complexity is easily justified as it gives users more control over their own data.

7 LEGAL AND ETHICAL ISSUES

The most relevant laws to zero-knowledge issues are those concerning data privacy. As privacy becomes increasingly rare, many nations have released data privacy laws which require companies to both treat sensitive user data with utmost care as well as to track only the data which as absolutely necessary for operation. The recent high-profile data privacy law known as the General Data Protection Regulation (GDPR) in European Union provides strict guidelines for data collection. Without informed consent, the only reasons user data may be collected is if it is necessary to fulfill legal obligations, to protect vital interests of a person, or to oblige contractual obligations with a person or other subject. The GDPR also specifies that any data collected about an individual must be accessible to that person should they query for it, and this data must be deleted if the individual makes such a request. In the context of zero-knowledge, the actual amount of data necessary for collection is often reduced by using zero-knowledge protocols. Since less data is collected, it is significantly easier for a company to comply with privacy laws such as GDPR.

The ACM Code of Ethics offers two specific sections related to privacy and confidentiality for computing professionals. Section 1.6 of the Code of Ethics, titled "Respect privacy," which describes

similar moral requirements as the legal ones presented by GDPR. A computing professional is morally obligated to make clear exactly what data is being used, to explain how it is used, and to make require informed consent before collecting any such data [1]. The Code of Ethics also specifies that only the minimum amount of personal information necessary should ever be collected. With zero-knowledge proofs, the minimum necessary amount of data is often decreased, so many systems could feasibly reduce the data footprint of its users.

Section 1.7 of the Code of Ethics explains that it is important to honor confidentiality as any private data that is necessary for system functioning is by definition private [1]. Using zero-knowledge protocols to reduce users' data footprints naturally reduces the risk of inadvertent data exposure and confidentially breaches. Since significantly less data is collected in the first place, there is less risk of it being exposed to others.

8 FINAL REMARKS

While zero-knowledge proofs are still relatively new in the world of applied computing, they offer many potential benefits in terms of secure code and data privacy. Despite the apparent complexity of zero-knowledge protocols, it can be fairly simple to implement a quick zero-knowledge protocol. In contrast to the code security issues observed in our implementation of a zero-knowledge protocol for the discrete logarithm, there are multiple black-box tools out there which provide protocols designed with secure coding in mind in order to protect against cheating provers, colluding verifiers, and other potential attacks against the protocol. As zero-knowledge continues to become more accessible and popular, users should expect to see an increase in the amount of control they have over their data.

REFERENCES

[1] The Code affirms an obligation of computing professionals to use their skills for the benefit of society. (????). https://www.acm.org/code-of-ethics
[2] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2087–2104.
[3] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. 1988. Everything provable is provable in zero-knowledge. In *Conference on the Theory and Application of Cryptography*. Springer, 37–56.
[4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive* 2018 (2018), 46.
[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable zero knowledge with no trusted setup. In *Annual International Cryptology Conference*. Springer, 701–732.
[6] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. 2019. Aurora: Transparent succinct arguments for R1CS. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 103–128.
[7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 781–796.
[8] Fabrice Boudot. 2000. Efficient proofs that a committed number lies in an interval. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 431–444.
[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 315–334.
[10] Jan Camenisch, Rafik Chaabouni, and others. 2008. Efficient protocols for set membership and range proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 234–252.
[11] Kostas Chalkias. 2017. Demonstrate how Zero-Knowledge Proofs work without using maths. (Sep 2017). https://www.linkedin.com/pulse/demonstrate-how-zero-knowledge-proofs-work-without-using-chalkias
[12] Richard Chang, Benny Chor, Oded Goldreich, Juris Hartmanis, Johan Håstad, Desh Ranjan, and Pankaj Rohatgi. 1994. The random oracle hypothesis is false. *J. Comput. System Sci.* 49, 1 (1994), 24–39.
[13] David Chaum, Jan-Hendrik Evertse, and Jeroen Van De Graaf. 1987. An improved protocol for demonstrating possession of discrete logarithms and some generalizations. In *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 127–141.
[14] Uriel Feige and Adi Shamir. 1990. Witness indistinguishable and witness hiding protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. Citeseer, 416–426.
[15] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The knowledge complexity of interactive proof systems. *SIAM Journal on computing* 18, 1 (1989), 186–208.
[16] Kashmir Hill. 2012. How Target figured out a teen girl was pregnant before her father did. *Forbes, Inc* (2012).
[17] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 839–858.
[18] Bernard Marr. 2018. how much data do we create every day? The mind-blowing stats everyone should read. *Forbes, May 21st, www. forbes. com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/, accessed July 10th* (2018).
[19] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. 2013. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 397–411.
[20] Carl Pomerance. 2008. A tale of two sieves. *Biscuits of Number Theory* 85 (2008), 175.
[21] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.
[22] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 459–474.
[23] Searnsy. 2019. Searnsy/ZeroKnowledge. (2019). https://github.com/Searnsy/ZeroKnowledge.git
[24] Nigel P Smart. 1999. The discrete logarithm problem on elliptic curves of trace one. *Journal of cryptology* 12, 3 (1999), 193–196.
[25] Yu Takabatake, Daisuke Kotani, and Yasuo Okabe. 2016. An anonymous distributed electronic voting system using Zerocoin. (2016).
[26] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 926–943.
[27] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. *IACR Cryptology ePrint Archive* 2019 (2019), 317.