# The design of zk-SNARKs and the improvements offered by Universal zk-SNARKs

Stephen Rowe
*Trinity College Dublin*
Dublin, Ireland
rowes@tcd.ie
Exam No: 45248
Student No: 14319662

***Abstract*—In this paper, I study the design of zk-SNARKs, presenting the mathematics behind the proofs in simplified, easy to understand manner. I introduce real-world applications of the proof system, in particular cryptocurrencies. I also measure zk-SNARKs against the more recent advancement of Universal zk-SNARKs, which aim to enhance the trustworthiness of the zero-knowledge proof system.**

## I. Introduction

zk-SNARKs, or "Zero-Knowledge Succinct Non-Interactive Argument of Knowledge" are a method of proving that a computation has been carried out correctly using a dataset, without revealing that dataset to any other party. We must introduce the roles of prover (party that carries out calculations) and verifier (person who must confirm or deny that the prover is telling the truth). While we may be familiar with public key signatures that can be used as succinct proof of the possession of static data, the issue arises that we may need to prove possession of dynamically computed data, and also be able to prove that the calculation was carried out correctly (or with no malicious manipulation). This difficulty has arisen in the world of cryptocurrency, as the consensus of the block-chain relies on peers on that block-chain verifying inputs from other peers, but the actual transactions between end-users should, ideally, be encrypted. zk-SNARKs are utilized in both the zCash and Ethereum cryptocurrency as a method of encrypting transactions in order to ensure the privacy of both parties in a transaction. As this is a very math-heavy topic to begin with, I will present a simplified version of the different steps taken to create, prove and verify a zk-SNARK with examples, in order to demonstrate the security and privacy of the system.

## II. The Logic behind zk-SNARKs

### A. Meanings of "Zero Knowledge" and "Non Interactive"

I'll break down the abbreviation of zk-SNARKs, explaining in plain English what each term means, before diving more in-depth into their implementation. The "Zero-Knowledge" [1] aspect relates to the fact that our prover must prove his knowledge or possession (of anything) without sharing any information about that knowledge/possession. In other words, he tells the verifier, "I've figured out the answer to this problem, you can't see any of my workings, you can't see any of my answers, but I can provide proof that it's all correct". "Succinct" refers to the proof being verifiable within a few milliseconds, being achievable with a small amount of data, perhaps a string, or in the case of Sonic (discussed later) in 256 bytes of data. There are two types of Zero-Knowledge proofs – "interactive" and "non-interactive", which is the next aspect of the abbreviation.

### B. Difference between Interactive and Non Interactive

An interactive proof is one that takes the form of what is essentially an interrogation – the verifier must constantly test the prover, and the prover must be right enough times such that the probability of him lying/cheating is negligible. Zero-knowledge proofs of this type can take any form; a simple example would be if you were colour-blind and I wanted to prove that I was not colour-blind. I hand you a red block and a green block. You hide the blocks behind your back, make the choice of if you are going to swap them in your hands once or leave them as is, then show them to me and ask which is the red and which is the green. If I get it right the first time, there is still a possibility that I was colour-blind and simply guessed correctly. So we repeat this experiment over and over until the probability that I was colour-blind and guessed correctly every time was negligible. This approach of interrogation (i.e. interactive proofs) is completely impractical in the case of a block-chain, as I would have to carry out this time-consuming interrogation with every peer on the block-chain in order to reach a consensus on the transaction that I want to add [2]. A non-interactive proof is much more desirable but is a massively more daunting task when we introduce the factor that calculations and parameters must also be kept secret. It requires that the proof be provided once by the prover, no interrogation is involved between prover and verifier, and the verifier can accept or deny the prover's evidence.

### C. Modular Arithmetic

The modular arithmetic and prime fields that we studied for AES encryption are also utilized in zk-SNARKs, as there are a number of advantages to this approach that make zero-knowledge verifiable proofs possible. As a quick refresher, we have a Finite Field $Z_p^*$, with $p$ being a prime, which consists of the elements $\{1,\ ...,\ p-1\}$ . This set contains a generator, $g$, which can be raised to some power $g^m$ to produce every

element in the above set, with $m$ being an element in the set $\{0, ..., p-2\}$. Just like with AES, we can take advantage of the Discrete Logarithm Problem, which states that, given a large $p$, a finite field $Z_p^*$, and an element $k$ in our Finite Field, it is very difficult to find an exponent $n$ for $g$ (our generator) that produces $g^n = k \ (mod\ p)$, meaning we can use this as a method of encryption. Using the finite field $Z_p^*$ allows us to use both addition and multiplication operators on our set. With all of this established, we can now move onto the advantages of using $Z_p^*$, modular arithmetic and raising a generator to an exponent that allow for our approach to zero-knowledge proofs. The first significant difference from AES is that no decryption takes place in the proof process. In our system we have two people interacting with each other, one party is what we'll call the "Calculator", and one is the "Requestor". Calculator has promised that whatever two parameters you send him, he'll calculate their sum and send them back. But Requestor doesn't trust Calculator, he doesn't know if Calculator will calculate the right output value, and he doesn't want Calculator to actually know the parameters he sends him. Seems like an impossible task, but fortunately there is a solution.

### *D. Blind Evaluation, secret parameters*

Let's take $E(parameter)$ to mean encryption of a parameter using $Z_p^*$ and generator $g$. Now let's create an example with $p$ as 59 and $g$ as 2. (I'll omit repeating $(mod\ 59)$ over and over in this part, just assume it whenever $+$ or $*$ takes place). Requestor decides to encrypt two values, 7 and 13.

$$E(7) = 2^7 = 10$$
$$E(13) = 2^{13} = 50$$

But what if we did $E(7+13)$? That's the same as $2^{7+13} = 28$, which is the same as $2^{20} = E(20)$. So now we have two encrypted parameters, and the result we're actually expecting back from Calculator to verify he calculated it correctly (i.e. $E(20)$). But there is something important to note here: $2^{7+13}$ is equal to $2^7 * 2^{13}$, which is the same as $E(7) * E(13)$. What this means is that, Calculator can use the fact that he knows that the parameters he is sent (that is, $E(7)$ and $E(13)$) are encrypted, he can fulfil his promise of adding the two values together despite not knowing what they really are by calculating $E(7) * E(13)$ on his side then sending that back to Requestor. This process is called Homomorphic Hiding [5]. Requestor can confirm that this is equal to $E(20)$. Now let's ramp it up a bit; this time Calculator says, "I have a polynomial, $F(x)$, but I don't want you to know what the polynomial is. If you send me a parameter, I can calculate $F(each\ of\ your\ parameters)$ and return you the results". So Requestor will not know the polynomial $F(x)$ at all, and Calculator will not know the parameters $x$, he'll just receive it encrypted. The polynomial will be of the form

$$a_0 \ + \ a_1 * x \ + \ a_2 * x^2 \dots \ a_m x^m$$

with coefficients being $a_0$, $a_1$, $a_2$, ..., $a_m$. So let's multiply a parameter that our Requestor sent to Calculator by a coefficient. Let's create a simple example that Calculator's secret polynomial is "$f(x) = 6x$", and we want to pass $E(x)$ as the (encrypted) parameter. The result that Calculator produces should be $E(6 * x)$. This is equivalent to

$$2^{6*x} = (2^x)^6 = E(x)^6$$

(remember, our generator is 2). What we've discovered from both of these examples is that we $(i)$ have a method of addition with hidden parameters, $(ii)$ have a method of multiplication with hidden parameters, and as a result $(iii)$ have a method of evaluating polynomials with hidden parameters. In other words, we can blindly evaluate polynomials and linear systems. This will prove very valuable for the concept of zero-knowledge proofs.

We can advance this concept further with the introduction of what's called Tate Reduced Pairing, which involves Elliptic Curves [8]. I will not go into too much detail in this topic, but the underlying concept is this: if we have a group $F_p$ that is of order $m$, the elements of which are points on the elliptic curve that can be generated by a generator in $F_p$, and from this we create a multiplicative group $F_{p^k}$, then $F_{p^k}$ contains $(d–1)$ subgroups $H_1$, $H_2$... (with $H_1$ being the subgroup $F_p$) that are groups that can be generated on the elliptic curve, all of the same $d$-order. Then we can use what's called a Tate mapping in which we take a generator from $H_1$ and a generator from $H_2$ to find the generator for $H_x$, some other subgroup of $F_{p^k}$, such that

$$TateMap((genrH_1)^a, (genrH_2)^b) = (genrH_x)^{ab}$$

We don't need to know the implementation of this Tate mapping to understand zk-SNARKs, we just need to understand that, since $H_1$ encrypts a parameter $x$ as $E_1(x)$, and $H_2$ encrypts a parameter $y$ as $E_2(y)$, then $E_x(xy)$ can be calculated as $TateMap(E_1(x), E_2(y))$. This concept will appear later as part of the no-interaction part of verification.

### *E. Verifying beyond basic arithmetic*

From the last part, an issue arises – although we have a way to evaluate polynomials blindly, Requestor is left in the dark about if Calculator carried out the calculations correctly. He does not know the secret polynomial, and even if he did, there would be no point in asking Calculator to calculate it – Requestor may as well just evaluate it himself. So we need to introduce a system to throw parameters at Calculator and determine from the result if he actually carried out his function using our parameters. What Requestor does is called the Known Coefficients Test, in which he picks an element alpha from our Finite Field $Z_p^*$, and two elements from the Finite Field, $(a, b)$, that must meet the following criteria:

- $b = a * alpha$
- $a, b \neq 0$

We will call this our challenge pair $(a, b)$. The most important factor here is that we call $(a, b)$ an $alpha$-pair (that is, if $alpha$ was a value of 3 then it would be a 3-pair) due to our first rule above, that $b$ is a multiple of $a$ in the field. Requestor

sends the challenge pair, $(a, b)$ to Calculator, but doesn't tell them what alpha is, and requests another $alpha$-pair. The only way that Calculator can do this is by multiplying both $a$ and $b$ by some new non-zero value, $\omega$, since he doesn't know what $alpha$ is, and has no way of calculating it due to the Discrete Logarithm Problem. This results in another $alpha$-pair (that is, for the same $alpha$ value). So at this point, we have an $alpha$ value that Requestor chose, a $\omega$ value that Calculator chose, and a method for which Calculator can demonstrate that he knows a value for $\omega$ that produces another $alpha$-pair, which Requestor can verify with a simple equality check,

$$alpha * (\omega * a) = (\omega * b).$$

(This is based on the 3-Round Knowledge-of-Exponent Assumptions outlined by Bellare, Palacio, [10] which states "For any adversary A (i.e. our Calculator) that takes input $q$, $g$, $g^a$, $g^b$, $g^{ab}$ and returns $(C, Y)$ with $Y = C^b$, there exists an extractor $\hat{A}$ , which given the same inputs as $A$ (our calculator) returns $c^1$, $c^2$ such that $g^{c1} * (g^a)^{c2} = C$"). The value of this is that it shows the starting point from which we can prove that both $(i)$ Calculator carried out a calculation out using our parameters and $(ii)$ Requestor can verify that his parameters were used for the calculation. But at this point, this is a very simple and impractical usage, as we only have a challenge pair $(a, b)$ and one $\omega$ value, there is not much we can do with this, we must expand it. We can do so by adding more challenges at once, i.e. $(a_1, b_1), (a_2, b_2), (a_3, b_3), \ldots, (a_n, b_n)$, and we need $n$ $\omega$ values, $\omega_1$ to $\omega_n$, (i.e. our coefficients) so that Calculator can reply with a Response Pair $(a', b')$, where

$$a' = \omega_1 a_1 + \omega_2 a_2 + \ldots + \omega_n a_n$$
$$b' = \omega_1 b_1 + \omega_2 b_2 + \ldots + \omega_n b_n$$

From this, we can clearly see how Calculator can have their own secret linear combination/polynomial, $f(x)$, to which Requestor can throw parameters at, and determine from the response (without knowing the polynomial or calculating anything) if Calculator's results are valid and actually used our parameters.

So, if Requestor has a value of 5 that it wants to pass to Calculator's polynomial, and they know it is a polynomial of degree 4, then Requestor randomly picks an alpha value of 3, and sends the following $(a, b)$ pairs:

- $(E(5^0), E(3 * 5^0))$
- $(E(5^1), E(3 * 5^1))$
- $(E(5^2), E(3 * 5^2))$
- $(E(5^3), E(3 * 5^3))$
- $(E(5^4), E(3 * 5^4))$

This shows the relationships between linear combinations and polynomials in our system, as Calculator treats this as 5 distinct parameters $v, w, x, y, z$ in its linear combination, but Requestor knows that they are $x^0, x^1, x^2, x^3, x^4$, with $x$ being 5 in our case. Calculator can then evaluate the polynomial value, he simply passes each of the above $a$-values as parameters for a simple linear combination, as opposed to inputting an $x$ into $f(x)$. Calculator will evaluate this for the $a$-elements first, then the $b$-elements, and return both results to Requestor, who can again validate that the $a$-result and $b$-result are still $alpha$-pairs. [11] Up to this point, we have established

- A way to hide parameters
- A way to carry out algebraic calculations on parameters that are hidden from us
- A way to verify that a calculation has been carried out on hidden parameters without having to carry out the calculation.
- A way to achieve the previous point, while also not knowing what the polynomial that was executed was.

### F. Quadratic Arithmetic Programs

Quadratic Arithmetic Program, or QAPs, are a method used to translate a computation into a polynomial [4]. They are very useful in zk-SNARKs, as they can be used to succinctly summarize a computation and prove that a computation has been carried out. Their implementation is useful in our case due to the Schwartz-Zippel Lemma [3], which proves that if two polynomials do not "agree" on one point, then they are extremely unlikely to agree on multiple points. In simple terms, this means that if Calculator claims a solution for his calculation, but this solution does not match up with rules we define (in the QAP), then his calculation is certainly invalid. From a computation, we create a circuit based on the following rules:

- Each vertex is an operation, $*$ or $+$
- Each vertex has a Left and a Right input, and a single Output.
- We create labelled wires $\rightarrow$ one for each input value, and one for the result of each multiplication vertex (outputs of addition vertices are treated as one combined input to the next gate) $\rightarrow$ these make up our legal assignments.

For each of our multiplication vertices, we associate some number (from our Field). The set of these values for our multiplication vertices is called our "target points". We will design a set of Left Polynomials, Right Polynomials and Output Polynomials. The number of each is the same as the total number of legal assignments. This set of polynomials is essentially a summary of our circuit. Each of our legal assignments is numbered, $1, \ldots, n$. "Fig. 1" is a subsection

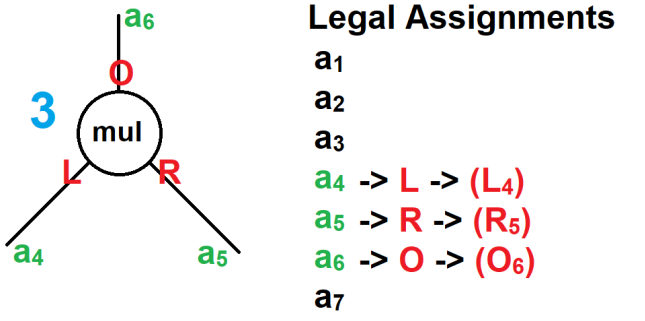


Fig. 1. Example of a multiplication vertex

of our circuit, focusing on the multiplication vertex that has been assigned a target value of 3. We can see from the table on the right which is the Left Input, Right Input and Output. We also see how to determine which polynomial that we need to focus on for each, so $L_4$ is one of the polynomials we need to focus on for this gate as $a_4$ is a Left input of the multiplication vertex. What this means is, since our target value is 3, $L_4$, $R_5$ and $O_6$ all must evaluate to zero for all target points except 3. In other words, the polynomial $L_4(x)$ must be zero when anything (from our field) other than 3 is passed as a parameter. I'll skip over working out what these polynomials should be, as all that we need to know is that they must meet the above criteria. So now we have legal assignments $a_1, a_2, \ldots, a_n$, and a set of polynomials, $L_s$, $R_s$ and $O_s$. From this, you will be able to see in "Fig. 2" that we construct polynomials $L$, $R$ and $O$. And finally, we create the Polynomial $P = L * R–O$, also demonstrated in "Fig. 2". Ignoring everything else for

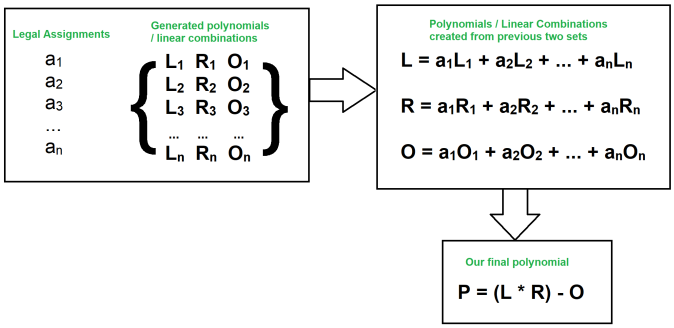


Fig. 2. Construction of polynomials

one moment, note that our polynomials $P$ is equivalent to $[SomeLeftInput] * [SomeRightInput]$ (which is what a multiplication gate calculates) minus the $Output$, meaning it should evaluate to 0 if the Output of a multiplication gate is correct. But in our actual polynomial $P$, we've calculated $L$, $R$ and $O$ as linear combinations of $L_s$, $R_s$, and $O_s$. Again, remember our rule that in our set of generated polynomials/linear combinations above, a given polynomial should disappear when passed a number that does not match the multiplication gate number that the polynomial is associated with. What this means is, when we pass a parameter 7 to $P(x)$, then only the polynomials from our sets $L_s$, $R_s$, and $O_s$ that are associated with multiplication gate 7 will remain for $L$, $R$ and $O$, with everything else evaluating to 0. Going back to the "Fig. 1", let's say we pass 3 as a parameter to $P(x)$. Since

$$L = a_1L_1 + a_2L_2 + \ldots + a_nL_n$$

and since we determined that $L_4(x)$ should be non-zero when 3 is passed as a parameter and all other $L_n$s should be 0 when 3 is passed, then the $L$ in our $P$ polynomial will be

$$a_1 * (0) + a_2 * (0) + \ldots + a_4 * (1) + \ldots + a_n * (0)$$

which evaluates to $a_4$. Similar for $R$ and $O$. Finally, since $P$ has the structure $L * R - O$, then this should also evaluate to 0, when the legal assignments are correct. From this we can see that with just the polynomial $P$, we can determine if the values $a_1$, $a_2$, … $a_n$ (that is, values which Calculator has determined), were valid.

### G. Target Polynomial

We've established that that the numbers assigned to the multiplication vectors are "target values", so from these, we can create a "target polynomial", $T(x)$. If our target values are 3, 10, 9 (chosen here at random), then our target polynomial will be $t(x) = (x–3)(x–10)(x–9)$. Due to the relationship between target values and the polynomial $P(x)$ that we established in the last paragraph, we know that our polynomial $P$ is then divisible by $T$, that is $L * R–O = T * S$, with $S$ being some polynomial, given that they have all been passed the same parameter. Again, the divisibility of $P$ by $T$ is only possible if the legal assignment values $a_1, a_2, \ldots a_n$ are valid. We have constructed our Quadratic Arithmetic Program, and consists of the sets $L_s, R_s, O_s$ and the polynomial $T$. Note that $P$ is not present in the QAP, as $P$ can only be calculated using the legal assignments, unlike everything else. Please see [9] for an alternative explanation of QAPs.

### H. QAPs in our zk-SNARK system

So again, lets return to the relationship between Requestor and Calculator. At this point, let's assume that the QAP is public knowledge, (that is, $L_s$, $R_s$ and $O_s$ and the polynomial $T$). Calculator wants to be able to say definitively, "You told me that I needed to follow a set of rules when calculating a value. I now have this value, and I can prove that I followed all the rules". Calculator wants to prove that he has a polynomial $P$ that meets the constraints of the QAP, which was constructed from legal assignments $a_1, a_2, \ldots a_n$, that prove that he has a solution to the polynomial given some input parameters (remember, $a_1, a_2, \ldots a_n$ are assigned from inputs, intermediate values (i.e. multiplication gate values) and outputs, including the final output from the polynomial). Let's consider this in an interactive situation – Requestor picks a parameter $E(z)$ and calculates $E(T(z))$. He then sends the parameter $E(z)$ to Calculator (remember from earlier, we have a method of sending hidden parameters that Calculator can work with). Calculator evaluates $E(P(z))$ and $E(S(z))$, with $S$ coming from $P = T * S$, i.e. the result of dividing $P$ by $T$. Requestor gets these values back, and can verify that $E(P(z)) = E(T(z) * S(z))$. If this equality holds, then Calculator has successfully proven to Requestor he has $a_1, a_2, \ldots a_n$ and has correctly evaluated the polynomial according to the predefined rules.

### I. Implementing Zero Interaction

At this point we have a reliable method for a party to prove that their calculations are valid without the polynomial or arguments being publicly known, but we still have an element of interaction that we must eliminate in order to meet the criteria of a zk-SNARK. For this, we must introduce the concept of the "Common Reference String", which is generated by

a trusted third party. We must return to both the concepts of $alpha$-pairs and Tate pairing that were mentioned earlier. It is the responsibility of the trusted third party to develop the QAP and what are called "toxic waste values", that is, the $alpha$ values, for which Calculator's calculations must be tested against. Fortunately, due to the Tate mapping, Requestor (i.e. the verifier) does not even need to know what the alpha values are, which eliminates the interactive requirement of proofs that we needed so far. Calculator calculates the pair:

$$(a, b) = (E_1(P(x)),\ E_2(alpha * P(x)))$$

using values specified by the trusted third party in the Common Reference String. We'll just call these two values $a$ and $b$ from here on out. Requestor (or any verifier) can accept or deny their proof with the calculate the values

$$a' = TateMap(a, E_2(alpha))$$
$$b' = TateMap(E_1(1), b)$$

keeping in mind that Requestor can extract the values $E_2(alpha)$ and $E_1(1)$ from the Common Reference String. Finally, an equality check can establish if the calculation was carried out correctly. As a result, we have finally met all of the criteria of a zk-SNARK – we can succinctly prove a calculation, using hidden parameters and polynomials, non-interactively. There are some things to note here however, the trust third party must generate these values (the common reference string) in an extremely secure way, as knowledge of the "toxic waste values" can lead to invalid proofs being created.

## III. Real World Applications and Improvements from Universal zk-SNARKs

### A. Implementation in Cryptocurrency

zk-SNARKs are the primary focus of the cryptocurrency zCash [6], and have more recently been implemented in Ethereum as of 2018. In the case of zCash, if Alice is sending Bob a sum of money, then calculations are based on Alice's spending keys and the amounts specified. As posession of the spending key proves authorization to spend, it is vital to prove possession of this key without the key being publicly known, and that the inputs and outputs from the calculations are valid and private. zCash achieves this through the creation of "Commitments", which are hashed values based on this data and a secret unique number for the transaction, which are essentially cheques on the block-chain, unspent transactions. The Commitment is confirmed by the spender through the use of a "Nullifier", a hash of the spending key and the previously generated transaction identifier, which propagates the transaction. The zero-knowledge proof aspect comes in with respect to proving all of the following:

- A Commitment exists relevant to this Nullifier
- The Commitment's calculations are valid

Thanks to the Hashing in relation to the private spending keys, it is infeasible for a malicious party to modify the transaction (without the possession of that spending key). In respect to the Calculator/Requestor dynamic I used throughout this paper, the Requestor role is played by any verifier on the block-chain, which shows the benefit of succinct proofs, as the same proof can satisfy any verifier.

### B. Advantages of Universal zk-SNARKs and SONIC

Universal ZK-SNARKs are an advancement on the procedure of generating a Common Reference String. The most significant issues with the previous approach were that the zk-SNARK had to be developed for a specific application (e.g. zCash would have to develop their own CRS specific to their system), the significant lengths that engineers would have to go to in order to ensure that the "toxic waste" generated in the creation of this Common Reference String was destroyed so that false proofs could not be generated, and the significant computational costs and complexity of generating these parameters. This would be generated in what is called a Ceremony, and you can see the zCash ceremony here [7]. Universal zk-SNARKs on the other hand generate updatable Common Reference Strings, such that only one CRS would have to be developed yet applied to a vast range of applications and would consistently be updatable by any party involved. The main goal would be to eliminate trust concerns in the initial setup, as trust is the basis of zero knowledge proofs, as the many potential applications of this Reference String is unknown, it is isolated from its uses. The Universal zk-SNARK system I will focus on will be Sonic, which was based on the work of Groth et al. and which requires only a single trusted setup that can be used for any application – only one ceremony is required. The main benefit of Sonic over typical zk-SNARKS is that at any time, a user may update the parameters, even after the system is live. One unique aspect is that if a genuine (trustworthy) user generates their own update to the Global Reference string, than the String is validly and trustworthy from that point out, that user can have complete confidence that any further updates must be genuine (although that user can't be 100% sure of the of validity of proofs generated from before that user created an update). Therefore the trustworthiness/validity is cumulative in favour of trustworthy and genuine users. Sonic also has the benefit of "Helpers", which is an aggregate party with a large amount of computational power, that calculates an advice string to speed up computations for verifiers, meaning that verification can be carried out in $[constant\ time] * [numberof\ proofs\ being\ verified]$. The Helper cannot fully be trusted, and should not be allowed communicate with provers, which limits this functionality to an extent, but it still drastically speeds up the verification process and reduces verifier's overhead. Proof sizes are as succinct as 256 bytes, which is another significant advantage of the Sonic system.

## IV. Conclusion

Throughout my research of zk-SNARKS and Universal zk-SNARKs, the term "moon-math" constantly cropped up in reference to the sheer complexity of the mathematics behind the

process, and the fact that you would intuitively think that what it achieves was near-impossible to begin with. zk-SNARK technology has developed massively in the last few years, and it was an insightful experience to study it at a time when the technology is still unravelling and practical implementations (in the form of cryptocurrency) are becoming more common. In terms of privacy concerns, the fundamental mathematical properties that zk-SNARKS takes advantage of for encryption, in particular with respect to the Discrete Logarithmic Problem, are well-beaten paths and are therefore trustworthy in that respect. Also, learning the actual mathematics of how it would be possible to blindly evaluate polynomials gave me great insight and trust in the zero-knowledge proof system, I feel I have gained a good grasp of the security in that respect. It was also very insightful to see the elaborate lengths the zCash engineers went to in order to ensure that the public parameters of their system were securely calculated, although it was alarming to learn that a Zero Knowledge proof system of any kind still has the fundamental flaw of being able to generate false proofs if the calculation of the initial parameters was retained. Saying this, seeing the further advancements of Universal zk-SNARKs as a method of defeating this weakness definitely promotes confidence in the technology as a whole. Future advancements in zk-SNARK technology will have the goal of reducing computation cost further, as generation of QAPs are a computationally expensive procedure, as well as reducing the computation cost of verification in the case of Universal zk-SNARKs, which at the moment are propped up by Helpers.

## REFERENCES


[1] Babai, L., 1990, July. E-mail and the unexpected power of interaction. In Proceedings Fifth Annual Structure in Complexity Theory Conference (pp. 30-44). IEEE.

[2] Kiayias, A., Miller, A. and Zindros, D., 2020, February. Non-interactive proofs of proof-of-work. In International Conference on Financial Cryptography and Data Security (pp. 505-522). Springer, Cham.

[3] Bishnoi, A., 2021. Two proofs of the Schwartz-Zippel lemma. [online] Anurag's Math Blog. Available at: ¡https://anuragbishnoi.wordpress.com/2015/03/17/two-proofs-of-the-schwartz-zippel-lemma/¿ [Accessed 24 May 2021].

[4] Gennaro, R., Gentry, C., Parno, B. and Raykova, M., 2013, May. Quadratic span programs and succinct NIZKs without PCPs. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 626-645). Springer, Berlin, Heidelberg.

[5] Zeroknowledgeblog.com. 2021. Homomorphic Hiding. [online] Available at: ¡https://www.zeroknowledgeblog.com/index.php/the-pinocchio-protocol/homomorphic-hiding¿ [Accessed 24 May 2021].

[6] Bowe, S., Gabizon, A. and Green, M.D., 2018, February. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In International Conference on Financial Cryptography and Data Security (pp. 64-77). Springer, Berlin, Heidelberg.

[7] Youtube.com. 2021. ZCash Ceremony. [online] Available at: ¡https://www.youtube.com/watch?v=D6dY-3x3teM¿ [Accessed 24 May 2021].

[8] Gabizon, A., 2021. Explaining SNARKs Part VII: Pairings of Elliptic Curves - Electric Coin Company. [online] Electric Coin Company. Available at: ¡https://electriccoin.co/blog/snark-explain7/¿ [Accessed 24 May 2021].

[9] Buterin, V., 2021. Quadratic Arithmetic Programs: from Zero to Hero. [online] Medium. Available at: ¡https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649¿ [Accessed 24 May 2021].

[10] Bellare, M. and Palacio, A., 2004, August. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In Annual International Cryptology Conference (pp. 273-289). Springer, Berlin, Heidelberg.

[11] Groth, J., 2010, December. Short pairing-based non-interactive zero-knowledge arguments. In International Conference on the Theory and Application of Cryptology and Information Security (pp. 321-340). Springer, Berlin, Heidelberg.