Corda and Zero Knowledge Proofs

Mike Hearn
mike@r3.com

December 19, 2019

Draft

Abstract

Integration of cryptographic zero knowledge proofs with Corda would allow peers to be convinced of the validity of transactions and thus the contents of the ledger, without revealing the transactions themselves. This privacy upgrade would be highly worthwhile but obtaining a production-grade integration of zero knowledge proofs requires solving several research-grade problems first. This short white paper examines the challenges involved in bringing such a feature to launch, proposes some potential solutions and compares the approach with one based on utilising secure hardware enclaves.

Contents

1 Introduction

Zero knowledge proofs. Algorithms for the creation of *succinct arguments of knowledge* without revealing private inputs (ZKP algorithms) are an active research area, with advances being published in the literature at a rapid pace. Their appeal for blockchain/distributed ledgers is that they can convince users of the correctness of the data in the ledger, whilst hiding the evidence that would otherwise be demanded. *Correctness* in this context means that all database constraints have been enforced. In blockchains this is typically done via so-called 'smart contract' logic and may be responsible for simple rules like users being forbidden from creating money out of thin air, and also much more complex business rules governing inter-firm business dealings, such as distributing the insurance risk of a large contract portfolio over many intermediate reinsurers.

In this paper "ZKP" refers to algorithms which prove arbitrary arithmetic circuits, and which yield reasonably small proofs. The term can also refer to use-case specific algorithms (such as the CT algorithms in Bitcoin) or remote attestations from hardware enclaves, but in this paper we always mean techniques like Groth16 SNARKs[1] or Bulletproofs.

Corda. Corda is a production-grade decentralised database that powers some of the world's largest enterprise blockchain deployments. As a motivating example, the Marco Polo project is a peer-to-peer trade finance network with over 30 participating organisations, and there are hundreds of other Corda business networks being developed or going live at the moment.

Due to the size and rapid growth of the ecosystem, techniques for applying ZKPs to entire classes of Corda applications simultaneously would have large business and social impact. On the other hand, individual per-app 'ad-hoc' techniques are less attractive, as a typical business software development team does not have any experience with advanced cryptography, nor do they want to learn it. A large part of what makes Corda useful is that it packages advanced cryptographic, peer-to-peer and blockchain technologies in a way that Java-speaking domain specialists find approachable. In Corda even tokens are just an app like any other: there is no native currency or coin. This is to allow a wide variety of token approaches to be developed by the community, for instance, tokens representing different kinds of assets, interest bearing or non interest bearing and so on.

For that reason this paper will take as given that any integration of ZKPs into Corda will happen at the platform level and be as transparent as possible to application developers. Although complete transparency is not required, approaches that would imply very large developer retraining or app rewrites are not considered.

Although this paper focuses on Corda we believe the same issues and challenges would apply to any enterprise-grade DLT.

Hardware enclaves. Corda's current strategy for upgrading privacy is to execute contract logic inside Intel SGX™ enclaves, by embedding a small JVM inside it and running bytecode processed and sandboxed by a determinism rewrite pass. This yields a standard programming model and only small levels of performance loss, as well as supporting renewability in case of compromise (see §3.11). So why plan to go beyond that and integrate cryptographic zero knowledge proofs?

The essential driver is that as a hardware based scheme, by design SGX cannot protect data against invasive hardware-level attacks. An adversary capable of decapping a chip and extracting a CPU specific key using a scanning electron microscope can defeat SGX security in ways that cannot be fixed (beyond simply making newer chips more physically tamperproof).

The extremely small transistor sizes used in modern chips combined with SGX-specific physical circuit obfuscation means this sort of attack is considered very difficult for any non-government level adversary, and has yet to be demonstrated in public. Corda explicitly excludes well funded governments from its threat model because its users are typically under the jurisdiction of those governments anyway, so this level of (end-game) security is good enough. The history of prior secure hardware efforts in smartcard and games console security also provide a lot of information on how the threat landscape is likely to evolve. However, physical attacks get steadily cheaper with time. Standard cryptographic problems have shown impressive long term resistance to being solved. A proof tactic based on well studied problems of known hardness would therefore be highly desirable (but see §3.10 for discussion of the significant caveats hiding behind this statement). As a secondary consideration, reliance only on ordinary arithmetic would insulate the platform from vendor roadmap changes.

2 Intended audience

This version of the paper assumes an understanding of modern zero knowledge proof technology at a high level, some compiler theory and how Corda itself is designed. Future versions may provide further explanatory information.

3 Challenges to be solved

To bring ZKPs to production in Corda there are many challenges which need to be solved. They can be categorised as either technical or social.

Technical challenges include:

1. Building a JVM bytecode arithmetising compiler.
2. Just-in-time circuit generation.
3. Building and updating a common reference string.

4. Building key management infrastructure.
5. Solving limits on recursive proof composition.
6. Incremental rollout into live systems.

Social challenges include:

1. Standardising circuit-friendly cryptographic algorithms through industry accepted forums.
2. Algorithmic agility: establishing a process for evaluating and migrating to new algorithms.
3. Building trust in stable algorithms.
4. Switching to 'standard' hardness assumptions.
5. How to recover trust in the ledger after algorithmic or implementation failures.

Each challenge will be briefly described in one section.

3.1 Bytecode arithmetisation

3.1.1 Overview

ZKP algorithms typically require the statement being proved (the program) to be encoded as an arithmetic circuit: a collection of addition and multiplication 'gates' analogous to an electronic circuit. If x and y are in {0, 1} then boolean logic can be expressed this way:

```
NOT(x): 1 - x
AND(x, y): x * y
OR(x, y): x + y - (x * y)
XOR(x, y): x + y - 2(x * y)
```

Due to the functional completeness of NAND gates this is sufficient to express any pure function, although there are often more efficient ways to encode things than using binary logic.

Corda applications express rules for relational integrity using JVM bytecode. Database entries (called 'states') contain constraints over what class files are acceptable, thus allowing a form of code upgrade. Any JAR file (zip) containing Java bytecode which satisfies the constraints of all states read or written from transaction may be attached to that transaction. The state contains the name of a class in that JAR file at which execution begins.

The platform first performs a static analysis and rewrite pass to enforce fully deterministic execution on the code to be executed, which includes imposing Ethereum-style opcode quotas, blocking access to sources of non-determinism like clocks, and preventing various tricks that may try to break out of the

sandbox. Then a method on the contract code is executed with a deserialized object graph representing the transaction. If the method runs to completion the transaction is deemed acceptable, if it throws an exception the transaction is considered invalid. Thus a Corda smart contract is a pure function.

Corda uses a subset of Java bytecode and not something more restrictive because data update rules in real business applications are often complex. A typical mid-tier server in a centralised web app is responsible for enforcing data validation rules that are frequently too difficult for even SQL driven RDBMS engines to handle, but in a decentralised and peer to peer system there is no trusted mid-tier to enforce data integrity. Corda users fully exploit this capability and frequently do things like big integer maths, string manipulation (e.g. to check IBANs), compare large object graphs and other tasks that demand the full power of an industrial strength language. Of course this design is far from unique: blockchain systems have included bytecode interpreters of varying strengths since Bitcoin 0.1.

All existing integrations of ZKP with blockchain platforms require experts to hand-craft circuits for the business logic of the application. The flagship ZKP-integrated ledger is arguably ZCash, which doesn't support smart contract logic for private tokens at all. Artisanal circuitry on a per-app basis is unworkable for Corda given the very large number of apps available for the platform, the complexity of the business logic they implement and the tight budgets with which they're developed.

For this reason we need a way to translate JVM bytecode to arithmetic circuit form automatically.

3.1.2 Constraint compiler

Circuit form has many limitations and requirements that make such a transformation difficult. Nonetheless, it can be done and compilers for other languages have been built before. Thus in some ways this is actually one of the easier challenges to solve: it's just a matter of engineering.

To convert bytecode to circuit form (and from there to polynomial constraints) a process similar to just-in-time compilation can be used. The following description assumes some familiarity with sea-of-nodes intermediate representations used by compilers like Graal and HotSpot C2 work. The reader is referred to Dubosq et al, *'An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler'*[2] for an introduction.

The following conversion algorithm is conjectured but gives a flavour of what would be required:

1. The program is processed by the deterministic Java rewrite subsystem outlined above. Amongst many other things, this ensures the code that follows will not be multi-threaded.

2. The program is converted to *static single assignment* form, in which all variables are assigned only once. In places where the program requires a variable be overwritten, e.g. a loop counter, special *phi* nodes are inserted that act as a switch.

3. Program inputs (e.g. arguments to the translated method) are mapped to variables at the start of the method and their positions recorded. These will be attached to 'input wires' of the resulting circuit. Reference inputs are replaced with allocations of the corresponding object type (but constructors calls are not emitted), and an additional wire representing whether the input was null.

4. All method call sites are either devirtualised completely or replaced with an exhaustive polymorphic inline cache, and then recursively inlined. Zero knowledge proofs have no notion of dynamic program loading, so, this kind of transformation under the closed-world assumption is possible. Calls to some special methods that have efficient pre-written circuit translations ('gadgets') are substituted, as is done with compiler intrinsics. We now have a graph of nodes. The graph now represents the entire program, with no method calls remaining.

5. Escape analysis is run over the program graph. As there are no threads and the entire program graph is visible due to exhaustive inlining, no allocations should escape and thus all should become eligible for scalar replacement. Scalar replacement decomposes object allocations to a set of individual variables for each field. A description of both optimisations is provided by Stadler et al[3]. In cases where a reference variable mapped as an input is scalar replaced, all the scalars are also mapped to input wires recursively. In this way the object graph passed to the program is converted into an array of group elements (integers modulo the order of the finite field of the circuit).

6. Due to the exhaustive scalar replacement performed in the prior step, all variables in the program are now primitive types or arrays of primitive types. Range analysis is done over all variables to establish integer range constraints. Statically inferred bounds may potentially be combined with profiling data and developer annotations to sharpen the ranges. Array bounds must likewise be inferred or provided by the developer, allowing arrays to be replaced with sets of individual variables representing each slot. This is likely to require a fairly sophisticated propagation pass so that e.g. bounds on the internal arrays used to encode `BigInteger` objects make it through the encapsulating objects.

7. Control flow is now progressively eliminated, such that every arm of a branch is executed and the corresponding phi nodes are replaced with selector constraints. Circuit form has no notion of control flow, so all loops are unrolled or exploded up to their statically determined bounds. Any loop induction variables that could not have range constraints inferred

during the prior step trigger an error and the developer is requested to provide a 'reasonable' bound, as a bound of the maximum range of an integer would yield impractical circuit sizes.

8. The prior steps may individually not always succeed, as they can depend on each other to be fully effective. Thus the process is repeated until a fixpoint is reached. If any control flow or allocation nodes still exist at this point, the program could not be reduced to a circuit and an error is signalled.

At the end of this process the program should be in a form only slightly higher level than an arithmetic circuit and can be progressively lowered via a sequence of passes until the final circuit is obtained. The circuit can then in turn be converted to a rank-1 constraint system (R1CS form), which acts as the input to the next step of the proving pipeline.

3.1.3 Implementation in an open source compiler

The Graal compiler infrastructure contains some of the transformation passes described above, along with many other production-grade program optimisations. It natively represents programs in SSA form and so it would be most appropriate to adapt it. Unfortunately, some enhancements would be required to enable the rather unusual transforms required:

1. Loading arrays with non-constant indexes triggers materialization of the array, i.e. scalar replacement will fail. Fixing this requires implementing stack allocation of arrays or some ZKP specific transform.
2. Objects are materialized if their type is checked.
3. Objects are materialized if a control flow merge joins two separate types.

Additional code would need to be developed to avoid object materialization in these cases.

Optimisation of the compiler itself may also be required, in case some of the transformations don't scale to the large IR graphs involved.

3.2 Just-in-time circuit generation

The variety of smart contracts Corda accepts is larger than practically deployable with any current ZKP algorithms. This is for several reasons:

Performance ZKP algorithms prove programs many orders of magnitude more slowly than normal code execution. Simple programs may frequently require 30–60 seconds to prove on powerful hardware, and more realistic programs can take hours.

Compatibility Due to the complexity of the arithmetisation transform, some programs will not be automatically translatable until the developer either provides additional hints or simplifies their logic.

Range restrictions Circuits require code topology and thus loop bounds to be known ahead of time. It may often be difficult for developers to really know what their range constraints should be: for instance, developers may find it hard to predict how many database objects a user may wish to combine into one transaction. If they guess too large their circuit will grow dramatically and thus performance will suffer or become impractical, but if they guess too small, some transactions will be unable to be proven with the generated circuit.

All ZKP algorithms are sensitive to circuit size. If a circuit gets too large it may no longer fit in memory, may start requiring compute clusters to work with, or simply have unusably poor performance. Additionally because circuits require fixed control flow topology the size of a circuit corresponds to the maximum *worst case* execution and frequently cost of evaluating the circuit is also the worst case cost. For example, if a collection could conceivably contain 10,000 elements, then the cost of looping over that collection is equal to just repeating the code inside the loop 10,000 times. Some techniques such as the 'energy saving' trick used in Geppetto[5] can reduce the cost, but it's still very significant. For these reasons even relatively simple business logic as used in ZCash can require a large circuit: the ZCash circuit contains around 2^{17} multiplication gates alone, and more when addition gates are considered.

Combined with the need for incremental deployment (§3.6) this suggests a form of partial ZKP usage, in which transactions and programs that fit the constraints of the technology use it and those that don't fall back to other techniques like secure hardware enclaves, or simply revealing the transaction data to the peer. Circuits that fit certain 'data shapes' could be created on the fly by the platform, based on real-world profiling data. If a collection could conceivably contain 10,000 elements but normally contains only 3 (a realistic example) then Corda would notice this and produce a smaller, more specialised circuit that can only handle up to three elements. If a transaction is found that doesn't fit, a fallback to other techniques would be automatically triggered.

Just-in-time circuit generation is required for other reasons. Developers need fast feedback loops when developing their app and adjusting their business logic. In some cases adjustments to business logic may constitute an emergency situation, for instance, to comply with a sudden regulatory deadline or the discovery of a serious bug. Additionally it is standard practice for developers to use continuous integration in which business logic is stressed with many complex tests, and developers are likewise sensitive to the feedback times from their CI systems. Exploding integration testing overheads or app deployment times is unacceptable.

Finally, there are many different ZKP algorithms that represent different trade-offs. For instance, Bulletproofs doesn't require any trusted setup procedure (see §3.3) and makes relatively standard cryptographic hardness assumptions (see §3.10) but doesn't scale as well as other algorithms that do.

Thus it would be useful for the platform to profile transaction traffic to determine which data shapes and contract logics are best paired with which algorithms (or not proven at all). This process would be controllable via developer hinting and operator commands, to allow for testing of ZKP dependent codepaths. This is especially important because it's common for Corda applications to distribute public or semi-public data, thus hiding these transactions would be pure overhead, with no actual increase in privacy.

3.3 Building and updating a common reference string

Many ZKP algorithms require a *common reference string* (CRS): a dataset generated from some randomly chosen values and then published. The random values must be destroyed after usage, as knowledge of them allows forgery of proofs. This critical requirement poses problematic logistical issues.

A recent line of work has developed algorithms like Sonic[6] and AuroraLight[7]. In these zkSNARKs the CRS may be created by many parties working together. The CRS is secure as long as *at least one* contributor to the CRS is trustworthy. These breakthroughs significantly enhance the practicality of deploying ZKP to production.

There are some practical details still to manage. A CRS is only trustworthy if you can reason about the likelihood of (complete) collaboration between the contributors. Being able to reason about this requires you to know the identities of the contributors. In a totally open system this isn't possible as a CRS may have been created by many Sybils of the same person. Corda networks (sometimes called 'zones') require some form of identity to be issued to nodes. Any form of identity can work including easily forged or stolen identities such as email addresses. However in the global Corda network operated by the Corda Network Foundation a relatively high standard of ID verification is used, in which company registrations are verified and the employment of the correspondent agent with that company or organisation is checked. Whilst not reaching banking know-your-customer standards this level of ID verification is good enough to allow users to conclude that at least some of the contributions for any given CRS were legitimate (including for strings used in the past, when the group may have been smaller).

Some CRS constructions are either per-circuit or per-circuit-size (usually, up to a circuit size). This means that there may be a desire to phase in new reference strings over time as hardware improves and larger circuit sizes become practical. Signalling protocols will need to be developed, along with distribution infrastructure and tooling. For instance, tools would be required so zone administrators can view the progress of building the new CRS and as the CRS set in use is a consensus critical parameter, Corda's network rules consensus mechanism (called *network parameters*) would need to be extended so everyone can be brought into alignment on which strings have had enough contributions to be considered trustworthy.

Finally, it would make sense for nodes to automatically contribute entropy to the CRS building process, without requiring administrator intervention. This would effectively shift questions of liability in case of problems to the Corda developers, and avoid an extensive hearts-and-minds programme with node operators who may otherwise not understand or want to take part in the process.

3.4 Key management

Once a program is converted to a constraint system it is further processed into proving and verification keys. These are analogous to the public and private keys in a conventional asymmetric cryptosystem, but both keys are public. The proving key is combined with per-proof public (visible) and private data to generate the proof. The generated proof is then combined with the public/visible data and the verification key to test whether the statement represented by the circuit was true. Crucially, the verifier doesn't need the private data which is what makes these algorithms zero knowledge. Public data may seem initially useless but in fact is required, for instance, it will contain the ID of the transaction being verified so the verifier can know which transaction was being proven. It will also often contain so-called 'non-deterministic advice' generated automatically by the proving process, as part of various optimisations.

Whilst proofs may be very small in succinct schemes the proving and verification keys can get very large, as their sizes are related to the size of the input circuits. Key sizes in the tens of gigabytes range are not unusual.

Circuit shapes. A Corda network may require many circuits to be in use simultaneously. This is partly because Corda contracts can be upgraded and may call into each other, for example, a bond contract may require that a payment of cash tokens is made atomically with an update of the bond's maturity data, the act of deciding how large that payment is may itself require invocation of logic to lazily apply interest calculations to a token and so on. As a consequence and because all logic an app invokes must be contained within the circuit for that app, releasing a new version of a CorDapp may impact many different circuits including those generated for dependee applications. Another reason for circuit churn may be upgrades to the proof infrastructure itself. If a new circuit compiler or proving algorithm produces more efficient constraint systems then it may be worth regenerating proving keys even in the absence of application logic changes.

Although universal circuits are theoretically possible, they come with extremely large overheads that render them of purely theoretical interest. It seems likely that custom per-shape circuits will be used for the forseeable future.

All this implies that Corda nodes may frequently need to generate, transfer, store and load files of the sort that are too large to practically fit inside the relational databases nodes use for storage today. Corda's current protocol is

optimised for transmission of small business messages. Thus a part of the upgrade work required would be to:

- Add support for large file streaming between nodes, including prioritisation and backpressure management to avoid causing unacceptable latency increases for sensitive business traffic during key transfers.
- Add support for local large file storage, possibly customisable for site-specific requirements. For instance nodes running popular cloud services have access to cheap object stores, but on-premises nodes may require additional support.
- Implement garbage collection of proving keys, so once a proving key has fallen out of use and only verification remains the space may be reclaimed. Old verification keys may also stop being used with time, as there is no equivalent to Bitcoin's reverification of the entire blockchain from scratch in Corda. In this case such keys may be archived to cold storage using a service like Amazon Glacier.
- Upgrade node resource management to ensure that attempts to load large circuits for processing don't overload a node. Most likely this means proving operations will be run in a separate microservice linked to the node via its built-in message queue broker. The MQ broker also provides many features that can simplify various operational aspects, such as load balancing, pushback, large file streaming and automatically dropping proving requests when the proving services are under heavy load. This may trigger fallback to hardware enclaves, for example.
- Define a data format that describes a **circuit shape**, suitable for use as a key in blob stores so circuits satisfying particular combinations of app versions, infrastructures and algorithms can be looked up and talked about on the wire.

Nodes must also be using the same keys, which leads to the question of who generates them. Ideally a node could download a circuit and use it, but how can the node know the circuit was correctly generated? For at least some deployments and algorithms this implies nodes must generate their own sets of keys from a described shape, and those keys must all be compatible. To what extent ZKP algorithms make this easy is a topic for future research.

3.5 Limits on recursive proof composition

A common task for a prover is to prove that an input zero knowledge proof is valid. This need to recursively compose proofs crops up in a variety of places, for instance:

- It allows proofs to be modularised, and thus for proofs to "call" other proofs. This in turn can help keep circuit sizes tractable.

- It allows for the proof of a validity of a ledger transaction to incorporate the proofs of the transaction's inputs, thus allowing a single proof to express the correctness of a transaction graph up to that point. As Corda uses the UTXO model like Bitcoin this is a powerful way to compress the transaction graph and avoid leaking private data through graph shapes.

Whilst recursive proof composition is not strictly required for deployment into Corda, it would be nice to have. This is because otherwise nodes must store the proof associated with each transaction ID, along with the inputs of those transactions, and then transfer the graph of all the proofs to each peer. This is no problem architecturally as it's essentially how Corda works today (albeit the 'proof' is the full data of a transaction). However composing proofs could improve Corda's scalability and performance properties, as well as making incremental deployment more tractable (see below).

Unfortunately current proof composition techniques have limited depth, that is, the deeper the chain of proofs-of-proofs gets the weaker the security becomes. Resolving this problem would be beneficial to deployment, but ultimately not required.

3.6 Incremental deployment

Any ZKP integration into Corda will need to be usable in a partly deployed state. This is because synchronous global upgrade of a live peer-to-peer network isn't possible. Additionally, replacing data transmission with proof transmission is a change to the consensus rules and thus would - in the base Corda architecture - require a global upgrade, no differently to any other blockchain system. A future version of Corda may allow nodes to outsource transaction verification to remote enclaves in case they fall behind a global upgrade schedule and such a feature might make consensus rule changes made after that point more tractable, but it would still eventually require a global coordination to avoid simply doubling the attack surface (from enclave compromise to enclave compromise + proof forgery).

As part of rollout, existing transactions in each node's transaction store will need to be processed and proofs generated. This is because in Corda there's no global data visibility: instead each node caches the dependency graph of each transaction they were involved in for later relay to other peers. Thus there's no way to calculate a universal proof of the state of the system up to that point from everyone's perspective, as nobody has the entire dataset, not even Corda's equivalent of the miners (a notary cluster).

Eventually the transaction backlog may have been processed in the background if nodes are sufficiently well provisioned but the system may need to 'fail over' to non-ZKP techniques at any moment, even after the initial upgrade. Some situations that require fail-over may be:

- A new circuit shape appears e.g. usage patterns change and thus some

fields become too large to verify, until new circuits are generated with larger unrolled loops or an app upgrade occurs (see §3.4).

- It becomes necessary to transmit a transaction graph to a peer for which not all proofs have been calculated, and for which the deadline is sooner than the estimated ETA of the proofs. A typical scenario in which this could happen is if there's a bulk import of data to the ledger: some users load millions of transactions into the ledger at high speed as part of migrating to their Corda based solution. This is possible in our architecture because creating states on the ledger doesn't require the involvement of notaries (miners), so it's limited only by the throughput of the node. The only way to feasibly handle this is to lazily calculate proofs on demand whilst the backlog is processed, but, if users wish to begin using the system before the backlog is processed and if transaction deadlines are faster than a proof can be calculated, fail-over must occur.
- Hardware failure causes loss of proof production capacity. This should probably not trigger a node outage as businesses may often prefer to take a slight loss of privacy over a shutdown of business, or they may prefer to fail-back to hardware enclaves.
- An algorithmic flaw is found that creates doubt about the veracity of proofs. The network operators wish to switch to other techniques until the flaw is analysed and fixed. Of course this can work in both directions: ZKP may be kept in backup and other techniques may be 'failed over' to ZKP if there is e.g. a bug discovered in enclave hardware.

In some cases a fail-over may need a change in consensus rules. Predicted changes to the consensus rules can be smoothly rolled out on 'flag days' using Corda's already existing network parameters mechanism. This can be used to force people on or off zero knowledge proofs depending on the decisions of the governing body of the network. In the Corda Network this is a democratically elected board of directors.

Infrastructure for switching back and forth must be created, tested and tooling for monitoring the state of the network must be created. In cases where fail-over may occur, users must understand what this means for their privacy and how to configure their node for their preferred availability/privacy tradeoffs.

3.7 Standardising circuit-friendly algorithms

Transaction proving circuits must be able to verify digital signatures. Embedding a classical elliptic curve or RSA verification algorithm yields very inefficient circuits, and thus a range of works have developed alternative algorithms that are more amenable to proof embedding. If migration to ZKP implies migration to new signature algorithms, as it probably does, then this creates several new additional pieces of work:

1. Selection and standardisation of an algorithm. Corda has some relatively

conservative customers who prefer an algorithm to be blessed by a recognised standards body before deployment e.g. IETF, NIST. DJB or Google are often acceptable also.

2. Addition to Corda, which is a consensus rule change requiring two-phase global rollout (may take years in a large network). Presumably, all transaction signatures must be switched to the new circuit-friendly algorithm before any circuits are used.

3. Something must be figured out for older transactions, for instance, additional circuit shapes that take into account the differing signature algorithms.

4. Ideally an argument that the new scheme is a post-quantum algorithm would be supplied. Most cryptography teams in large enterprises expect the next major algorithmic migration after elliptic curves to be to a post-quantum algorithm and would be uncomfortable adopting new algorithms (with their decade-long deployment timeframes) if there was no resistance to quantum computers.

A more complex migration may be required if the hash function also needs to change. Transaction IDs are calculated as the root of a Merkle tree over transaction components using SHA-2. SHA-2 is relatively expensive to encode in circuit form. Changing how transaction IDs are calculated would be a complex logistical endeavour requiring extensive testing and which may break backwards compatibility with applications.

3.8 Algorithmic agility

New ZKP algorithms are developed at a steady rate. It is fair to assume they will change with time. This leads to a requirement that new algorithms can (efficiently) verify proofs generated with older algorithms. If this capability isn't present then it would require a global re-proving of every transaction in order to switch to a new algorithm, something that may be logistically impossible given that most nodes will not have large numbers of idle cores just sitting around.

Verifying proofs for older or very different algorithms is not a topic that has yet been explored by the research community.

3.9 Building trust in stable algorithms

Cryptography is deployed into mainstream computing quite slowly. It took several decades for RSA and elliptic curve cryptography to become widespread after their development: for most developers, Bitcoin was their first exposure to *any* elliptic curves.

This level of trust is baked into how people think about cryptographic algorithms: as old, trustworthy, very well studied and correspondingly unbreakable.

This mentality doesn't map well to the reality of zero knowledge proof algorithms, which are frequently only a year or two old, rely on vastly more complex mathematics than used in any widely deployed cryptosystem, which are exposed to very small (and hard to measure) amounts of peer review and which have been broken several times.

Getting to the point where ZKPs are deployable on live Corda networks, which can handle hundreds of millions of dollars per day, will require a widely agreed upon algorithm to emerge. Then it will need to be standardised by trusted cryptographic authorities (which will be difficult given they may lack specialists in ZKP technology), and then many years must pass whilst peer review builds up. It would help enormously if the quantity of peer review an algorithm has received is publicly measurable, for instance, if cryptographers who read and study a paper assert their approval in some database. Current academic practice is that negative results aren't published, which may significantly increase the time required for industry to begin trusting an algorithm.

This problem is also fundamental in another respect: it assumes relatively slow algorithmic progress. Between the `secp256r1` curve being standardised and `Ed25519` becoming widely used nearly 20 years passed. The ZKP world has no such stability, with algorithms frequently being obsoleted within months. It's unclear when or if progress in this field will ever slow down enough for people to build trust in any single algorithm.

3.10 Switching to standard hardness assumptions

As discussed in the introduction, the primary reason to switch to cryptographic ZKPs is the belief that well studied 'hard' problems in mathematics have a better chance of resisting attack than physical objects. For example, the elliptic curve discrete logarithm problem (ECDLP) has seen minimal progress in over 30 years despite enormous advances in computational power. The RSA problem has become easier to attack as hardware got faster but can still be secure when suitably large keys are used.

Unfortunately, many zero knowledge proof algorithms are built on non-standard assumptions about the difficulty of novel mathematical problems. Non-standard assumptions aren't automatically an issue as with time some may become considered 'hard' via social consensus building. But it must be considered that the well deserved reputation ordinary encryption algorithms have for being impossible to break doesn't automatically translate to zero knowledge proofs. These new techniques are *developed by the field of cryptography* but are not standard cryptography.

Koblitz and Menzes examined the rapidly increasing number of novel hardness assumptions in their 2010 paper, *'The Brave New World of Bodacious Assumptions in Cryptography'*[8]. This concern is tricky to state precisely as (like in most academic fields) cryptography doesn't publish negative results, and thus terms like 'standard' and 'hard' are inherently subjective. Whether a problem

meets this criteria is effectively a function of time and adoption, leading to a catch-22 in which a problem can't become standard unless it's received lots of cryptanalysis, it doesn't receive lots of cryptanalysis unless it's widely adopted, but it can't be widely adopted unless it becomes standard.

As a motivating example, consider the Groth16 scheme[1]. This is probably the most widely deployed zero knowledge proof algorithm via its use in Zcash, and tends to be the default choice for other blockchain systems due to its tiny proofs and best-in-class performance. This SNARK is based on the *knowledge of exponent assumption* (KEA). The mathematical details of this problem don't concern us here; suffice it to say that it has received little study relative to more famous problems and therefore professional cryptographers lack confidence that it's truly hard. One reason for this belief is that the KEA comes in several variants, one of which was successfully invalidated by Bellare and Palacio[9] (but only after it was used to construct a peer reviewed cryptosystem). Another reason is the structure of the KEA makes it tricky to actually falsify at all.

Some ZKP algorithms do rely on standard assumptions and advertise that as a feature, but at the time of writing they all sacrifice efficiency to obtain that robustness. Given the serious performance challenges all ZKP algorithms face it's not surprising that most developers prefer the known and measurable gain in performance over the unknown and unmeasurable risk inherent in making unusual assumptions.

3.11 How to recover trust after failure

Strongly related to the questions of algorithmic agility and hardness assumptions is the challenge of system recovery after a flaw in either an algorithm or implementation is found.

Corda's design for deployment of secure enclaves has two phases. In phase one (called the *attestation model*) enclaves sign transactions to indicate validity, and these signatures combined with static remote attestations form a (hardware based) zero knowledge proof. If the enclave is compromised then its signing key might be leaked, e.g. via side channel attacks, and that would allow an adversary to prove arbitrary untrue things about the contents of the ledger, allowing them to (for example) forge tokens. Although phase one will be brought to production, we strive towards phase two (called the *privacy model*) in which peers exchange encrypted transactions and verify them locally. In this model if an enclave is compromised you may leak private transaction data to a peer, but nobody can break ledger integrity. This is a more robust and more trustworthy model given that private data gets stale and uninteresting with time, thus privacy leaks are in a sense self-healing. But integrity, once lost, cannot be recovered.

Unfortunately zero knowledge proofs only allow an equivalent of the attestation model. There's no equivalent to the privacy model (this would need something closer to multi-party computation). This means trust in the ZKP algorithms and infrastructure must be absolute: there's no way back from failure.

It's possible there is no solution to this problem.

Bibliography

[1] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 305–326. Springer, 2016.

[2] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[3] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM.

[4] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[5] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE, 2015.

[6] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Report 2019/099, 2019. https://eprint.iacr.org/2019/099.

[7] Ariel Gabizon. Auroralight: Improved prover efficiency and srs size in a sonic-like system. Cryptology ePrint Archive, Report 2019/601, 2019. https://eprint.iacr.org/2019/601.

[8] Neal Koblitz and Alfred Menezes. The brave new world of bodacious assumptions in cryptography. *Notices of the American Mathematical Society*, 57(3):357–365, 2010.

[9] Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Annual International Cryptology Conference*, pages 273–289. Springer, 2004.