Leaping the Innovation Chasm by Securing the Data

By Lou Senko
Senior Vice President and Chief Information Officer, Q2

A unique use of blockchain provides a radical approach to secure applications and the data itself, building a bridge across the innovation chasm. Blockchain-derived storage technologies can increase security posture while reducing the traditional friction and risk that is so often associated with the adoption of new technologies in highly regulated environments.

Q2

Leaping the Innovation Chasm by Securing the Data

By Lou Senko, Q2 Senior Vice President and Chief Information Officer

As competition and customer demands increase, the cycle times from code-to-customer has shrunk at a startling pace. This contraction means production teams are ingesting new software at a much higher speed, and more of their infrastructure is becoming software, operated as code. Much of that software is now open-source, "paint still wet," downloaded last week, and brand-spanking-new. It is a considerable shift away from the tried and true technologies that crawl along steadily, versioning year-over-year, a situation in which staff can take five-day boot camps and become 'certified' on the latest features. Currently, the learning trajectory is steep, and the innovation speed is high, but the experience operating these technologies is low. As good as production teams are, they are not perfect. Looking back at some recent newsworthy data breaches, such as when nation state actors entered Equifax's unpatched Apache Struts[i] or CapitalOne's misconfigured WAF[ii] allowed a former AWS employee to exfiltrate data using AWS' complex metadata services, teams paid a considerable price. Instead of stifling innovation with layers of controls and processes, double-checks, and bureaucracy, successful teams are leaning into an overhaul of their security posture to prevent these kinds of attacks. And they're not leaning in with incremental improvements of a gear update or refresh, they're reframing their approach with more radical tactics. Too often, security and compliance are viewed as opposing forces to innovation and speed, but in an oxymoronic way, more security and resilience leads to faster innovation. Less friction and less risk offer a window to try new things.

I am living this challenge with you. In my role as CIO at Q2, I am responsible for a team that delivers a high-level customer experience that ensures the availability, performance, and quality of our services exceed our customers' high expectations. Because we deliver this experience through our hosting environments, I am also responsible for those environments' compliance and security. We host the digital branch of ~450 financial institutions, supporting 17 million end users and moving over $1.5 trillion annually. One in 10 online banking users in the U.S. logs onto our platform, and we power a third of the top 100 banks in the nation. The team has evolved from being viewed as a high-executing, compliant, and metric-oriented *barrier* between our development team and our customers to become an *enabler* of getting the code to customers faster. This shift has been transformational not only for my team but also for our customers and the organization.

To describe the start of our journey, we took a very layered and common approach to improve our security posture. We began with the hosting environment's perimeter, using a typical Gartner Magic Quadrant, congo-line, best-of-breed, next-generation firewalls, advanced web application firewalls (AWAF), a slew of real-time threat feeds, and a SIEM triggering 100+ runbooks as we started adapting to the MITRE ATT&CK framework. Next, we moved into the typical system hardening, strict user provisioning, and a new Endpoint Detection and Remediation (EDR) deployment. After hardening the Hosting environments, we took a pass to implement a zero-trust model across our networks and doubled down on user endpoint hardening with a slew of new tools, including UAM (User Activity Monitoring) and EDR to protect against a user introducing the threat. Still, we did not know what we did not know, and learning from our mistakes at this level was a luxury we could not afford. All this was still not enough.

So, we took a **radical leap.** If the data is both the target and the risk point, what if we simply removed it? That's right; *the best way to keep the data safe is not to have the data at all*. What if we just assumed the network was already breached and the bad actors were already inside? How would we keep the data safe? How could we

increase our visibility and control? We explored adopting a security posture that moves the security into the application and into the data itself using tokenization and blockchain-derived storage technology.

Blockchain became famous as the immutable public ledger system underneath Bitcoin, tracking the currency's movement from entity A to entity B to entity C. Blockchain added the much-needed trust to the new unknown cryptocurrency. Is there a similar public ledger system coming for the financial industry? I hope so. When? I have no idea.

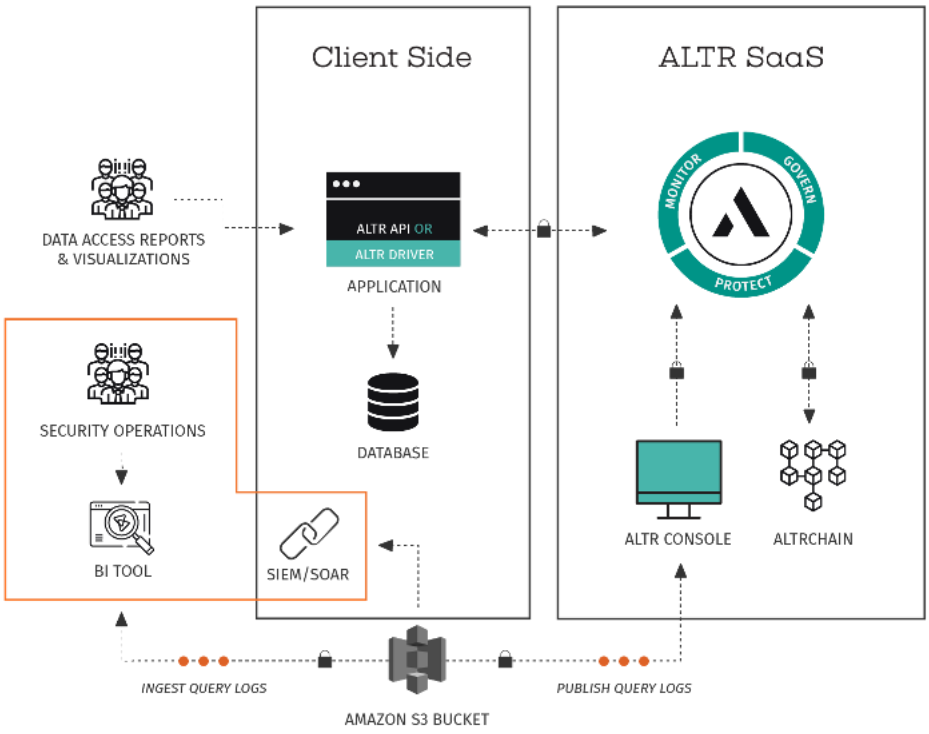
So, the security use case for blockchain technology is slightly different. Think about an average business, who has direct access to the money. Who can take from the petty cash box, write a check, or has account authorization? The answer is very few people. It is a very controlled process requiring many approvals. But who has access to the data in that same business? Often, lots of folks.

Encryption has been security's answer for a long time, but encrypting data still means the source data is there. You may not be able to read it yet, but you have it, and with some time and some math, you should eventually be able to get it. Encryption was not enough, as if you suffer a breach, you still have to report that the source data was stolen, even though encrypted.

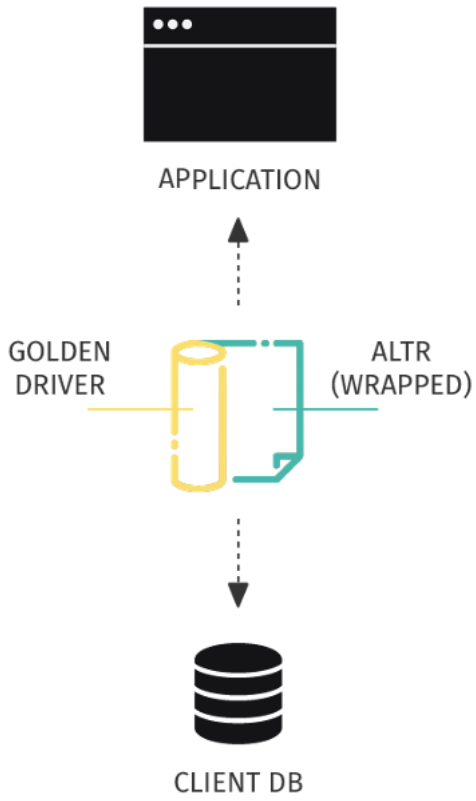
Under that lens, the approach leveraging blockchain technology is similar to how our financial institution customers protect money:

- Provide a **View** into how money is flowing in and out of the account
- Provide a **Valve** which can meter out the flow of money (ATM limits, wire transfer limits)
- Provide a **Vault** where no one person can access the store of money

There are three parts in a high-level review of the actual technology: a smart driver, the management gateway, and then, the blockchain-derived storage.



A smart driver sits between the application and the database where all data flows through it. The driver exposes a standard JDBC or ODBC interface that is fully compliant with applications and databases. The driver intercepts the data we have identified as valuable to protect it and requests a token from the platform to use in place of the identified useful data. What is placed back into the application's local database is simply a token and a pointer back to the management gateway – not to the data, not to the data's location, but to the management gateway itself as a reference. This is a giant leap forward in securing applications as we have essentially disassociated sensitive data from my application and the access that application provides to users while still maintaining end user functionality. An illustration of how the smart driver is implemented can be found below.



It should be noted that the manufactures supplied driver is instantiated by the driver wrapper, meaning the database manufactures driver still makes the underlying connection to the database server.

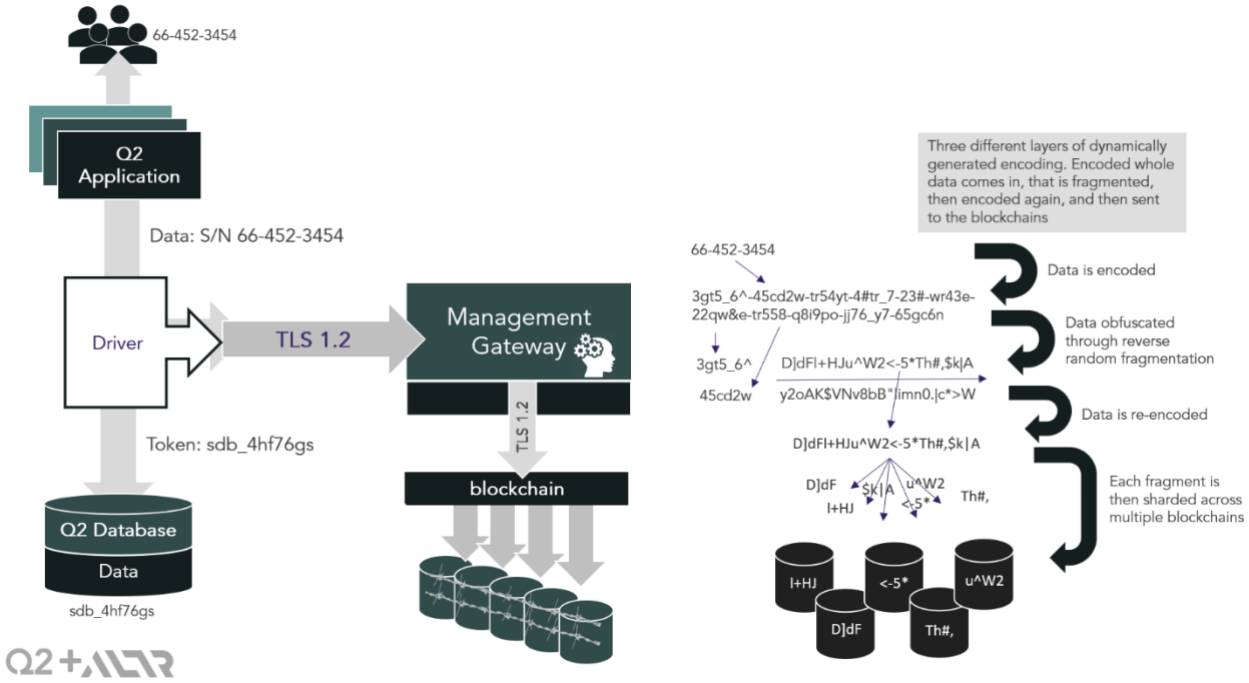
Once the valuable data hits the management gateway, the secure storage process begins effectively transforming the original sensitive data into many fragments of “new data.” The management gateway uses a patent issued process to encode and fragment the plain text data dicing it into random bits via reverse fragmentation. Each bit is then sent to the blockchain network, which sees this new data coming and transforms it again, for the third time. The information fragments are then distributed across multiple blockchains (the Vault). The Vault is a private, distributed ledger-based technology that has been optimized for high read/write performance, in the tier below the database tier, with no API surface to interact with, only a TLS connection to the management gateway itself.

The fragmentation and encoding process use industry-standard algorithms and protocols at run time in memory, which prevents any static mappings of data fragmentation. This method prevents a key from being stolen, which can be used to unlock data and prevent a product source code breach from "seeing" the encoding format and reverse-engineering it.

Each query sent to the database server by the application is captured, including both the requestor meta information and as the returned result set and how the application consumed that result set. All of this is stored in a tamper-evident and tamper-resistant blockchain structure, providing unprecedented visibility into data movement (the view) without the drawbacks of production database logging or network-based logging. The driver also allows for the setting of limits on data access, including volumes and speeds, all of which can be limited per application user (the valve). Usage can be confined to ‘known’ levels and limit the misuse, should it occur. So, if it is not normal for a customer service agent to retrieve thousands of rows of data, actions can be

taken, including alerts, blocks, or injection of latency into the data retrieval to provide a form of "digital quicksand."

High-level view:



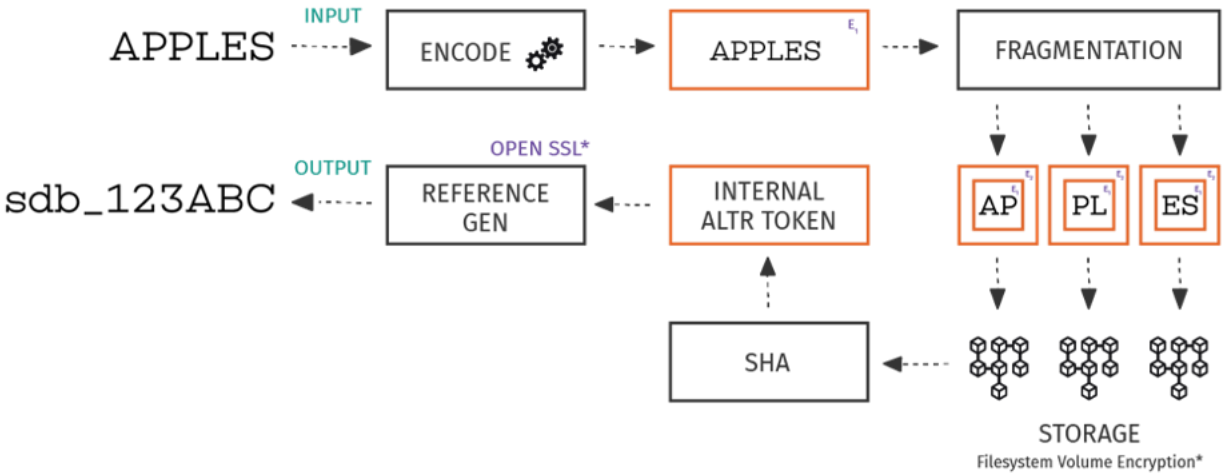
The technology used in this is from ALTR, an Austin, Texas-based startup, formed in 2014, launched in 2018, and comprised of experts in blockchain and low latency systems with a core team from the financial/trading industry. So, they are used to big dollars and high speeds. I will attempt to discuss the relevant pieces of the technology here.

A more in-depth technical lens:
Generally, the challenge is that traditional databases do not adequately protect against threat actors or internal resources, such as employees and technology staff, tampering with the data. At best, such systems typically provide audit access and modify the stored data, but the audit logs are generally mutable and can be changed just as easily as the data. So, an immutable datastore for low-latency reading and writing of large data sets is needed, with recent immutable examples of databases, including blockchain-based databases such as bitcoin and MultiChain. Blockchain systems are built upon ideas first described in a paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System,"[iii] and these systems typically implement a peer-to-peer system based on some combination of encryption, consensus algorithms, and proof-of-X, where X is some aspect that is difficult to consolidate across the network, such as proof-of-work, proof-of-stake, or proof-of-storage. Typically, those actors on a network having proof-of-X arrive at a consensus regarding the validation of peer-to-peer transactions, often using various consensus algorithms like Paxos, Raft, or hashgraph. Or some private blockchains do not implement proof-of-X consensus, e.g., where trusted parties control the computing hardware implementing the blockchain. Chained cryptographic operations tie a sequence of such transactions into a chain that once validated, is typically prohibitively computationally expensive to falsify. So, this version of blockchain serves as the trusted database and is described in detail later in this article.

While the data is in-flight, the solution uses an ensemble of data encryption techniques managed by HashiCorp Vault. The solution employs an AES256 symmetric key encryption process where the key is generated on-demand and sent through a separate communication path than the data itself. This process protects the data traveling between the components.

Once the data reaches a component in the solution stack called the management gateway (the brains), the data is streamed through AES decryption, fragmented into very small pieces that are encoded into a self-describing data storage format [iv] and sent to blockchains. At this point, the data fragments are no longer representative of the actual source data. If the data was originally PAN data subject to PCI oversight, it is now no longer PCI relevant due to its transformation.

To explain how a plain text value is stored and becomes a token, we can use the following diagram. The encoding steps are completed using industry-standard protocols, such as Protocol Buffers, for example. The structure of these buffers and the fragmentation indices are determined at run time, as mentioned above, preventing a static mapping. During the SHA step, the inputs into the internal token generation function include the position in the chain of the previously written fragment and a hash of all previously written data in that particular chain.

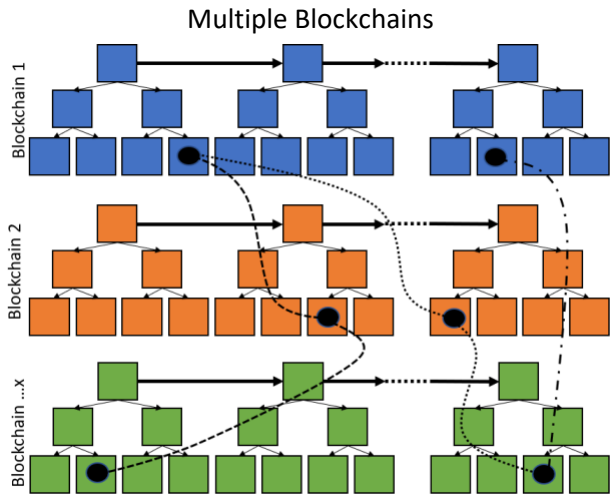


The data storage format includes *only* the encoded fragment of data and the blockchain address of where the following piece of data exists. The address of the piece of data that 'follows' the current piece of data is NOT relevant or valuable to the host where the data is stored. It is only valuable to the management gateway at the time that the gateway is given the correct input, as the gateway is entirely stateless; there is no one legend that shows a mapping of tokens to data, nor is there one map that describes where all fragments live, nor is there one virtual disk that would contain a single contiguous string of data.

Each data fragment in its self-describing format is sent to a different partition of the patented blockchains [v] for writing (the blockchains are designed [vi] to hold data as well as metadata regarding access – explained in more detail later in this article). This is achieved through an abstract hashing scheme [vii] that generates a cryptographic hash (presently SHA256) to assemble a Merkle tree data structure that gives the data fragments their immutable

properties. The hash of the leaf-node value in the Merkle structure where the initial fragment is written is ultimately what is returned from the management gateway and is then remapped to the token that the client receives [viii] for storage in their relational database.

Techniques exist for storing data in a distributed form in a peer-to-peer network, but such systems often present other problems. Examples include various systems built around the BitTorrent protocol. These systems often operate without a central authority to be compromised, but in these systems, when or if the data is altered (potentially to hide or obfuscate changes), there is no recourse or method to detect these changes. Further, many distributed peer-to-peer storage networks, like many implementing the BitTorrent protocol, maintain full copies of files (or other BLOBs) at each peer, potentially leaving the full file open to inspection by an untrusted party. To mitigate these issues and others, when writing data, a component referred to as an "arbiter" (which may be a piece of middleware and may be implemented in the translator) may be capable of taking as an input a fully formed datum (e.g., a unit of content, like a document or value in a database), starting from the last byte of the data, fragmenting that data into N. pieces (where N is an integer larger than 1, and in many commercially relevant cases larger than 5 or 10, and in some cases can be relative to the size of the initial data), and placing each piece on a physically (or virtually) separate blockchain-backed storage data structures, with each piece containing pointers to the next storage location of the fragmented data. The arbiter (or other middleware) then returns the TXID (or other node identifier) of the last written fragment (the first byte) to the application which requested the fragmenting of the data.



Fragmenting Data for the Purposes of Persistent Storage
Across Multiple Immutable Data Structures

The process of writing to the blockchain may include forming nodes of a directed acyclic graph having node content that includes the document, from a lower-trust database (the application datastore where we keep the non-sensitive data). Forming nodes may include accessing existing nodes and inserting node content into those nodes or forming entire new data structures by which nodes are encoded. Each node may include node content with a collection of attributes, which in some cases may include a pointer to another node. Those node attributes may include a pointer that is an identifier of an adjacent node in the directed acyclic graph, and a cryptographic hash value based upon one or more attributes of the adjacent node that is identified. These last two pairs of attributes (an identifier of another node and a cryptographic hash value of at least some of that node's content) may correspond to a cryptographic hash pointer from one node to another. Cryptographic hash

pointers may define edges of the graph, and an individual node may contain zero cryptographic hash pointers (such as a leaf node in a binary tree), a single cryptographic hash pointer (such as a link in a blockchain or linked list), a pair of cryptographic hash pointers (such as in a non-leaf node of a binary tree directed acyclic graph), or various other amounts of cryptographic hash pointers, for example, in splay trees or skip lists.

Then in reverse, when an application or resource requests the reassembly of fragmented data, an arbiter (or another piece of middleware) is supplied with the TXID (or other node identifier) of the first byte of the data. After reading the first byte, the arbiter or middleware then reads the subsequent pointers until a null character or end of sequence character is read. Once all the pieces have been read into memory, the arbiter or middleware responds to the application with the resultant unfragmented datum.

The obvious challenge is to overcome the latency required to reassemble and rehydrate the actual data at speeds that the application to tolerate and compensate for. Many extant blockchain-based databases are not well suited for certain use cases, particularly those involving latency-sensitive access (e.g., reading or writing) to large files (e.g., documents or other collections of binary data treated as a single entity, often called "blobs"), for instance in a blockchain-hosted filesystem. Indeed, many blockchain databases are not readily configured to store large objects and object files (e.g., on the order of 500 kilobytes or larger, depending on the use case and acceptable latency), as such systems are typically highly specialized for small-payload "transactional" applications. In such systems, when storing larger collections of binary data (e.g., files or blobs), the chain can dramatically slow as the chain gets bigger, particularly for write operations.

These and other problems are mitigated by a system referred as "Docuchain ." Docuchain is a blockchain software suite specifically designed for low-latency put and get operations of Binary Large Objects (BLOBs). Docuchain uses an improved version of an underlying data structure called "Merkle Trees" to provide the immutable properties of the blockchain. Docuchain is operative to respond to commands within less than 50 milliseconds (ms), e.g., and many commercial implementations are expected to provide sub 10 ms operations as perceived by the application interacting with the chain, and sub 5 ms operations at the level of the chain. Further, such response times are expected to scale with the chain size. Some may scale response times at nlog(n), wherein n is the number of entries in the chain.

Merkle Trees generally work by a tree of cryptographic hashes in which each element of the tree contains a hash of the information contained by its children, and leaf elements are hashes of the subject data stored in the Merkle Tree. In many traditional implementations of Merkle Trees for the purposes of storing data, like those in many earlier blockchain databases, the data is stored in a separate logical datastore, hashed, and just the hash is carried into the Merkle Trees. That is, the data being by the database stored is not part of the Merkle Tree; only a hash digest of the data is part of the Merkle Tree.

To mitigate some of the problems with traditional blockchain databases, Docuchain store the data directly in Merkle Trees. That is, when data is written to the database or read from the database, that data is written into specific fields of the elements (e.g., attributes of node content of nodes) of the Merkle Tree or read from specific fields of the elements of the Merkle Tree (rather than just a hash digest of the data residing in the Merkle Tree with the entire data residing in an external datastore). With this tight coupling, if the data is altered at all, the entire hash for the tree (as is distinct from a hash of a file stored in the database) is thrown off immediately, and ultimately the chain will be broken and detected as such during validation operations. The tight coupling of the

data to the tree is expected to reduce the number of read/write operations necessary to transact with the chain, further reducing potential latency.

As mentioned above:

- Docuchain contains two components: LibDocuchain and DocuSocket.
- LibDocuchain is a software library that contains the following data structures:
    * MerkleTree<H, T>
    * DataBlock<H, T>
    * BlockManager<H, T>

    - where "H" represents a cryptographic hashing function, and "T" represents the data type being stored in the chain. Generally, hashing functions map data of arbitrary size to data of fixed size. Examples of cryptographic hash functions include SHA256, SHA-2, MD5, and the like (e.g., applied more than once Docuchain, can use a variety of different types of hashing functions, and this is expected to include later developed hashing algorithms (e.g., new post-quantum cryptographic hashes).

The BlockManager is responsible for taking data, placing into the appropriate block and inserting into the appropriate Merkle tree, writing to disk at the appropriate times, increasing (e.g., guaranteeing) chain durability (i.e., tamper-resistance). Additionally, there is a JobManager that manages concurrent operations on the datastructures to deliver operation times.

DocuSocket is a software library that maintains connections to clients, e.g., remote computing devices seeking to access data stored in the database. Connections from clients may be relatively stateless (and in some cases, after a session is confirmed, there is no further state). A connection can accept an arbitrary number of requests in an arbitrary order and will return responses to the requests in arbitrary order. For example, the connection might receive write (A), write (B), read (C) operations, and the Docusocket may respond to request C before A or B. However, once a transaction response comes for A or B, those operations are considered final and are committed to the chain and are instantly queryable. This "always-on" and highly available connection is one differentiator that allows Docusocket to out-perform standard blockchains in comparable benchmarks.

Write requests are accepted through Docusocket and passed off to BlockManager for writing into a MerkleTree object. In some cases, the result of this operation is a hash composed from the position in the MerkleTree, the hash from the previous block, and the contents of the data that is written; this is called the Transaction ID (TXID), which is an example of a node identifier or pointer (which is not to suggest that these or other categories are disjoint). These TXIDs are stored in alternate data stores for later referencing and retrieval, e.g., in the lower-trust database in place of the data to which they point (like our application SQL database), or in a lower-trust file system in place of documents referenced by the TXIDs. In some cases, the TXID's are segment pointers to a linked list of segment pointers and segments that encode the stored data.

As noted, earlier blockchain databases can store blobs, but the way in which the data is stored and accessed often imposes severe performance penalties when accessing larger collections of data (e.g., larger BLOBs). Such systems often rely on additional functionality originally intended to handle the transactive nature of bitcoins and other cryptocurrencies. This additional functionality is often provided by an "in-chain" scripting language which defines the transactions that enter the chain. For many use cases, this is a secure and efficient method of maintaining a ledger; however, these features come at significant time complexity cost for larger chains,

particularly in write operations. As these chains grow, they become significantly slower. Blockchain document databases that store the documents in-chain are expected to grow very fast, thus putting significant limitations on what is practically possible with prior technology.

In contrast, because Docuchain store data in its exact representation (which should not be read to exclude various transformations and may include storing a compressed, encrypted copy of the data) directly in the Merkle Tree, those are expected to be able to circumvent the need to have the in-chain scripting language and can provide 0(log(n)) get/put operations on a tree once it is loaded in memory. Further, storing the data being committed to the database, rather than just a hash digest, in the Merkle Tree is expected to impede (and in some cases defeat) hash collision attacks on the stored data. In such attacks, malicious content is selected and designed to produce the same hash as stored data, and that malicious data is substituted in the database for the authentic data. With traditional systems, the Merkle Tree will yield a hash value that validates the malicious content as authentic. In contrast, you can circumvent this attack vector by storing the data committed to the database in the Merkle Tree. Further, you may detect changes without the need to continually verify each BLOB with its respective hash digest on a continual basis, unlike many systems that merely store a hash digest of the BLOB in the chain. That said, not all implementation will afford these benefits, e.g., as some may avoid the use of traditional in-chain scripts to access hashed digests in the chain, without storing the entire document, thereby expediting operations.

In summary, the use of this blockchain-derived technology provides an innovative, next-level way to not store the actual data to reassemble it on-demand and in real-time. The actual data only exists in memory while the application needs it, and it is never stored. Some other notable pieces:

1. All disks, including queues, databases, and most importantly, blockchain nodes where data is being stored, use disk-level encryption.
2. The solution encoding methods are not considered ‘simple encoding’ such as ROT13, which uses letter substitution replacing every letter with the 13th letter after it. For example, “test string” becomes “grfg fgevat.” This type of encoding only requires the data to encode and decode. The solution uses a schema-based encoding, which means you need the data + schema to encode and decode (similar to how you need data + key to encrypt). This way, someone can have the encoded data but still not be able to easily or quickly decode the data. The solution schema is baked into the application code, which resides on locked-down and secured Docker containers and is not accessible anywhere else on the network, meaning there is nothing you could intercept to decode the data (like you could intercept a key). This same process happens again in a different part of the stack after the initial fragmentation (the fragments are re-encoded), so we end up with multiple layers of protected, schema-based encodings for the data fragments.
3. Data is transformed through reverse random fragmentation. This fragmentation of the non-simple encoded data immediately makes the data fragments useless to any direct access. When the fragments are finally stored in the blockchain nodes, they are randomly sent to different blockchain networks. Each network is physically separated from each other, so access to one network does not give access to another network. The solution runs with a minimum of three distinct blockchain networks.

I share this use case as an example of a non-traditional use of blockchain technology used to improve our security posture as we house over seven petabytes of customer data within our hosting environments and must maintain a high level of regulatory compliance. This innovative approach has been validated by an ensemble of 26 patents (most relevant shared here) and several independent well-known PCI assessors. It has additionally withheld the scrutiny of an independent penetration test by a former 10-year National Security Agency (NSA)

engineer, in which he was unable to retrieve data from the system as well as a host of other large company security teams. Q2's new security posture (powered by this solution) recently won the 2020 CSO50 award.

As a result – sensitive data is not being stored in the database, only tokens. Sensitive data not being stored in the blockchain, only "new data" that has been randomly encoded three times, fragmented and scattered across multiple blockchains. So, it's not permission for us to leave a door to the hosting environment open, nor can we stop patching, but it does offer real next-level security for those teams trying new things and moving very fast, knowing someone is always trying to see if they can gain access to that valuable data.

---

[i] Department of Justice - Scott McCulloch, Trial Attorney – Counterintelligence and Export Control Section – Grand Jury Charges - https://www.justice.gov/opa/press-release/file/1246891/download

[ii] Department of Justice – Emily Langlie, Communications Director – U.S. Attorney's Office - Seattle Tech Worker Arrested for Data Theft Involving Large Financial Services Company - **https://www.justice.gov/usao-wdwa/pr/seattle-tech-worker-arrested-data-theft-involving-large-financial-services-company**

[iii] Bitcoin: A Peer-to-Peer Electronic Cash System – Satoshi Nakamoto – www.bitcoin.org – October,2008 - https://bitcoin.org/bitcoin.pdf#:~:text=Bitcoin%3A%20A%20Peer-to-Peer%20Electronic%20Cash%20System.%20Satoshi%20Nakamoto,to%20another%20without%20going%20through%20a%20financial%20institution.

[iv] United States Patent Application Publication – Pub. No.: US 2017/0364698 A1 - https://patentimages.storage.googleapis.com/38/ef/45/f2d0e327ec8480/US20170364698A1.pdf

[v] International Application Published under the Patent Cooperation Treaty (PCT): International Publication Number WO 2019/033088 AI https://patentimages.storage.googleapis.com/93/9a/87/4e426d80b98d2f/WO2019033088A1.pdf

[vi] United States Patent Application Publication – Pub. No.: US 2017/3064700 A1- https://patentimages.storage.googleapis.com/78/a5/ad/7435976d4ea1c7/US20170364700A1.pdf

[vii] United States Patent Application Publication – Pub. No.: US 2017/0366353 A1 - https://patentimages.storage.googleapis.com/16/d0/c0/a489dd0bc95671/US20170366353A1.pdf

[viii] United States Patent Application Publication – Pub No.: US 2018/0307857 A1 - https://patentimages.storage.googleapis.com/19/2b/42/f07f53c931b6b7/US20180307857A1.pdf