ENSO: A USER INTERFACE FOR THE DECENTRALIZED INTERNET

ALEXANDER COWARD[†]

Abstract. User interfaces accessible to nontechnical people are essential for the widespread adoption of any technology. In the case of standalone computers, a key breakthrough was the invention of the graphic user interface. As computers started to be connected to one another, another key breakthrough followed with the invention of the web browser. There is a need for a similarly bold step forward enabling ordinary people to interact with the decentralized internet, or web3. These new user interfaces need to be familiar enough that people do not feel lost or overly intimidated. At the same time, they must help people intuitively understand that the decentralized internet is fundamentally different to what has gone before, revealing for ordinary users the inherent superiority in the underlying technology. We hope our work in this document will form the basis of a new and better approach to how data storage and computation is communicated to users in all form factors.

1. Background and introduction

Let us start with a definition:

Definition 1. *Suppose Bob receives a message M. If Bob is able to independently verify the truth of proposition P using no information other than what is contained in M, we say that Bob has concluded that P is true in a way that is* trustless*. We will call the pair (M, P) a* certificate*. If a certificate is conveyed in digital form, for example using a telecommunications network, we call it a* digital certificate*.*

From the armchair, it is not obvious that there can be any certificates. When Bob receives the message, Bob has no knowledge of who sent the message, nor does Bob trust whatever communication system was used to convey the message to him.

In fact, even well before the invention of computers or modern telecommunications, people exchanged certificates. An example is a message adorned with a wax seal. The receiver of such a message is not able to independently verify the truth of the message itself, but they are able, by inspecting the seal, to conclude that whoever was in possession of the message at the time the seal was applied was in possession of the stamp that makes that particular seal.

It is important here that it be impossible to reverse engineer the stamp from the wax seal, and this explains the use of wax as the medium for the seal; it is both fragile and vulnerable to heat, and any attempt to make a negative impression, for example with molten metal, will immediately destroy the wax seal and produce no impression.

This example of using seals to stamp documents is entirely analogous to the use of public key cryptography on the internet. If a user visits `bankofamerica.com` in a browser they may very reasonably want to know that the website they see is indeed provided by Bank of America, and not an imposter. If this was done via documents sent in the mail, Bank of America could apply a wax seal made using its official Bank of America stamp. Today, they use RSA or some equivalent technology to sign the packets of data using a private key that corresponds to their public key. Analogously to the wax seal, it is impossible to reverse engineer the private key from the public key and signature applied to the message. Also analogously to the wax seal, a user is able to independently verify the sender of the message insofar that they can be sure the originator of the message or packets of data was in possession of something secret and special, either a stamp or a private key.

Despite this, a user still has to take certain things on trust when it comes to deducing the originator of the message. Most important, the recipient of a sealed or signed message has to know what a valid seal or signature looks like. If a recipient of a message purporting to be from Queen Elizabeth I has no idea what her seal is supposed to look like, how can they know if it is real or fake? This is analogous to the problem of matching a public key

[†]`hello@alexandercoward.com`

on the internet with a real identity of some sort.

The solution to this problem on today's internet is to bake into the browser[1] or operating system[2] of the client computer a list of trusted public keys. Because these trusted public keys and their respective owners are baked into software that is widely disseminated, the lists of trusted public keys are also widely disseminated.

This is analogous to assuming that everyone knows what the seal of Queen Elizabeth I looks like and trusts her not to lie about other people's seals. From there, if another person wants to verify their identity, they present a document that shows their seal has been vouched for by the known seal. For example, if the Duke of Norfolk wanted to send a message to someone and enable the receiver to verify that it was from him, he could send the message with his own seal, together with a message from Elizabeth I, signed with her seal, declaring: "This is the seal of the Duke of Norfolk."

Again, this is entirely analogous to the public key cryptography used in all modern web browsers. The root authorities are analogous to Queen Elizabeth I, and instead of signing a document that says "this is the seal of the Duke of Norfolk," they sign a digital certificate that says: "This is the public key of `bankofamerica.com`."

In practice, there is a lot more complexity to public key cryptography than what is described above. For example, the number theoretic cryptosystem RSA was not known until the late 1960s and not made public until 1977. Moreover, there are usually layers of delegated authority between the root certificate baked into a browser or operating system and the domain name a user visits. Certificates have expiry dates, which raises the thorny question of how the current time on the client is determined, and under some circumstances certificates may be revoked, with the signing of yet another certificate.

All this complexity is hidden from a regular user. Instead, the result of decades of research in applied cryptography is that the user sees a simple icon of a padlock. If the padlock appears, that says to the user: "This website is indeed the website indicated in the address bar." If the padlock is missing or has a line through it, that means: "This website might not be the website indicated in the address bar; you might be being tricked." Many ordinary users do not even have this basic understanding of what the padlock means, and simply interpret it as a vague sentiment of safety for whatever website they are visiting.

It is important that the padlock appears outside the rectangle of screen rendered by the website itself, as shown in Figure 1. If the padlock appeared on or over the rectangle of screen rendered by the website, then a dishonest website, for example provided by a man-in-the-middle imposter, could draw their own padlock.

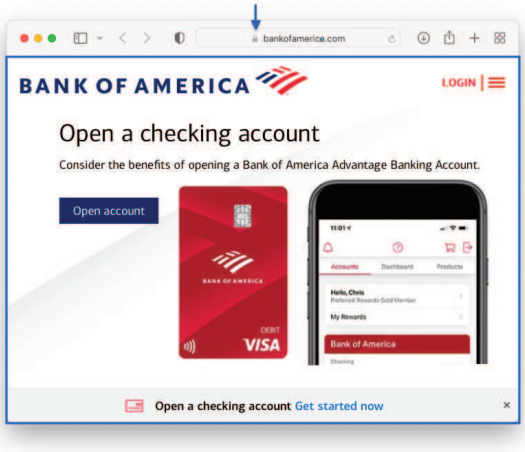

FIGURE 1. A padlock, crucially outside the rectangle of screen rendered by the website.

This is the current state of user interface design for normal users of the internet when it comes to the cryptographic soundness of the data they are viewing and interacting with. Provided the user trusts their browser or operating system to be judicious in its selection of root certificate authorities, the user may be sure that they are indeed visiting the website indicated in the address bar, and that is all.

The end user has no guarantees whatsoever that the website owner is not misleading or manipulating them in manifest different ways. A social media site can delete, distort or invent posts. A banking app can fabricate financial records. An email service can suppress email from competitors. Airlines can tell one user one price for a seat and another

[1]For example: `https://chromium.googlesource.com/chromium/src/+/main/net/data/ssl/chrome_root_store/root_store.md`

[2]For example: `https://support.apple.com/en-us/HT213464`

user another price. Beyond knowing that a website is indeed providing data from the owner of that domain, a user has no assurance as to the truth or falsity of anything they see or interact with, and no ability whatsoever to audit let alone change the processes that resulted in them being shown that information.

It hardly needs stating that this failure to provide end users with reliable trustless information is a central argument in favor of the decentralized internet, or web3. Today a sophisticated user can know in the true trustless sense that they have been sent the Bitcoin someone claims to have sent them. They can know trustlessly the state of a dApp on Ethereum. They can know trustlessly that a document shared via IPFS is indeed the document matching the CID for that document. They can know that the terms of a smart contract will be honored. They can know the price history of an NFT has not been edited for their eyes only. Moreover, they can interact with these systems secure in the knowledge that their wishes will be respected and not distorted. These are just some of the many benefits available to sophisticated users of the decentralized internet.

However most users of the internet are not at that level of sophistication, and cannot be expected to be. What is needed is a suite of user interface motifs that ordinary users recognize as legitimate and which app and dApp developers adopt with the same ubiquity that people have adopted the padlock to indicate that a website really belongs to the url in the address bar. Elaborating what those motifs must look like and how they must interact with and connect with existing and emergent systems is the topic of this whitepaper.

2. What is the decentralized internet?

The history of the internet is nearly as old as the history of the computer, and early pioneers began sketching out the basic principles of the internet at least as far back as the early 1960s [2]. In the time since then, there has been, in broad terms, an ongoing tension between two competing mindsets. The first mindset places computers in the hands of large institutions, and these computers are powerful by the standards of the day. The second mindset seeks to make computers available to ordinary people, with smaller physical size, enhanced usability and reduced cost.

At first glance, today it seems that the second mindset has won the day. Whereas in the 1960s and 1970s computers were found in the offices of banks, government agencies and other suitably resourced organizations, today not only do ordinary people own desktop and laptop computers, they also often own phones, tablets, watches and other devices that are essentially computers but in another form factor. Computers are literally all around us.

However, in parallel with the move to more ubiquitous computer ownership, since the early 2000s and the rise of "cloud computing" many of the benefits of computer ownership that led to the adoption of the personal computer in the first place have been lost. We have gone from accounting records being kept on a mainframe computer at a bank in the 1970s, to those records sitting on a user's own computer in the 1990s and early 2000s, to that same accounting data sitting on the servers of companies like Intuit today. The mainframe computer has returned by the back door, and it has done so in every industry.

The decentralized internet is an effort to reverse this tide and bring about a new era of personal computation, where not just computers but computation is genuinely in the hands of ordinary people. It can be regarded as being in the same spirit that companies like Apple sought to put a computer in every home and on every desk, not just in offices and banks and government buildings. This spirit is captured in the famous 1983 photograph of Steve Jobs[3] shown in Figure 2.

Let us consider a simple application, a to-do list, and examine various architectures.

In the "mainframe" architecture, before the internet, one could imagine a location on the high street where people go to maintain their to-dos in exchange for a fee. When someone thinks of a new to-do item, they go to that location and tell a person at the desk they want to add a new to-do. Similarly, when they complete an item they go along and ask for that to be recorded in a similar way. All the to-do data is kept by the company operating the location on a central computer at that location. If someone wanted to move to a different provider of to-do services they would have to open an account with another provider.

[3]Image courtesy of OSXDaily.

FIGURE 2. Computers, and computation, should be in the hands of ordinary people, not just giant organizations.

In the "application software" architecture of the early personal computers, before web browsers or when they were in their infancy, to-do software might be available in stores. A user could go to a shop and buy a physical disk, say a floppy disk or CD-ROM, which contained application logic for managing to-dos. The application logic and the associated user interface is installed on the user's personal computer. The to-do data is kept on the hard disk of the user's computer. If the user wants to make a backup copy of their to-do data, they can save it to something like a floppy disk drive or an external hard drive.

In the "thin-client" architecture that is common today, a user opens a web browser and creates an account at something like `todos.com`. That exposes a similar interface to the software that could be bought in the store, but instead of the user's to-do data being stored on the user's computer, it is stored on the servers of the company operating `todos.com`.

None of these architectures are satisfactory. The "mainframe" architecture where people visit a physical location is too slow for most purposes. Meanwhile, the "application software" and "thin-client" architectures take opposite extreme views as to how to handle user data. In the former case, the user has custody of their data, but making it interact with other software or users is not straightforward. Suppose, for example, that someone wanted to enable a family member to see their to-dos. That would mean that family member also had to install the same to-do application on their machine, which they might not want to do. Even if the family member did install matching software, we run into the issue of how to maintain a matching data structure for the to-do lists on each machine. Establishing consensus between the two machines shows how even by thinking about a very simple application, we quickly run into the need for a protocol to maintain consensus, a *consensus protocol*.

This need to maintain a consensus for application data explains the appeal of moving that data to a remote location that clients can connect to and use as a single source of truth.

However, users pay a heavy price for using a web application such as `todos.com`. When users hand over their data to a company, they hand over all control over that data. If the company is hacked, private information about the user can be leaked. If the company wants to make extra revenue, it can sell advertising based on user data, or the data itself. If the company goes bankrupt or changes its mind about what kinds of services it wants to offer, the service may cease to exist, and a user may lose their data altogether. Alternatively, a company may exploit high switching costs associated with changing provider and increase the price for using `todos.com` while offering no extra value.

In an ideal world, users would experience the benefits of application software, specifically the fine grain control over their data and the knowledge that the service will not abruptly change, while at the same time the rich interoperability and shared experiences provided by a well developed web application.

This is the vision of the decentralized internet. Rather than the browser acting as a "thin-client" where a user essentially makes a virtual visit to a company branch containing a mainframe computer, instead software providers reveal software that comes to the user's computer. Users then have tight control over where their data sits. For some types of data, such as accounting data they use for preparing their tax filings, they may prefer that it only sits on their personal machine. For other types of data, such as social media posts, they may prefer it to be publicly broadcast. Finally they may

prefer to share some data with specific other users, such as family members or their accountant.

While appealing, in the early days of the internet the technology did not yet exist to make this vision a reality. Specifically, hard problems related to consensus protocol design had not been solved, meaning that applications could not be kept in sync with each other without relying on central servers for each application.

The advent of Bitcoin in 2008, following decades of research going back as least as far as the early 1960s, for the first time showed that a robust data structure could be maintained with no central server. Just as early mainframe computers found application in the finance sector, so too Bitcoin has been regarded, falsely, as a financial innovation when in fact it is a theoretical computer science innovation. It is not just financial data that can be kept securely in sync with itself with no central server, but any data, including user application data. In the nearly 15 years since Bitcoin emerged, much further progress has been made, and we now have the ability to build applications that provide the best of both worlds, as shown in Figure 3.

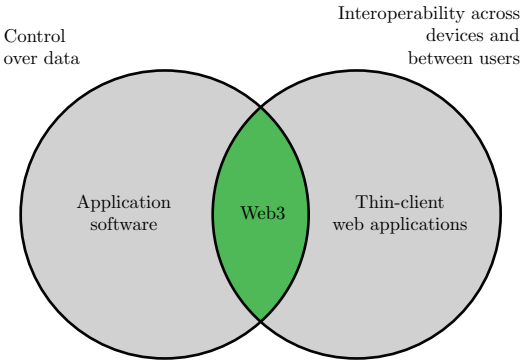

FIGURE 3. Built using blockchain technology, the decentralized internet, or web3, allows the best of both worlds.

Moreover, an architecture where application software runs on client machines and shares user data seamlessly without the need for a central server opens the possibility for a new generation of applications that have not been possible until now in any architecture. That is because once there is consensus about the state of an application, there is no need for there to be a single interface connecting to that data on the frontend, and users can use applications that read and write data in a way that is unopinionated about how other users are interacting with that data.

In summary, the best way to understand the decentralized internet is, for those of us old enough to remember, to think back to the days when we bought software in a store and kept track of our own data using our own storage. The decentralized internet is much more similar to that model than the browser based "thin-client" model. The difference is that now those applications can be richly and securely interoperable between users, devices and applications themselves without the need for a central server. Explaining how this magical technology works is the subject of the next section.

3. How the decentralized internet works

It will be helpful to recall that our intention of building applications that do not rely on trusting a central server is not at all new. In fact, we can recall the words of Paul Baran who introduced his seminal 1964 memorandum [2] as follows:

> "Let us consider the synthesis of a communication network which will allow several hundred major communications stations to talk with one another after an enemy attack. As a criterion of survivability we elect to use the percentage of stations both surviving the physical attack and remaining in electrical connection with the largest single group of surviving stations. This criterion is chosen as a conservative measure of the ability of the surviving stations to operate together as a coherent entity after the attack. This means that small groups of stations isolated from the single largest group are considered to be ineffective."

With these thoughts in mind, it becomes clear why a centralized architecture is vulnerable. If all communications go through a central headquarters it means that if that central organizing unit is destroyed all further communication disintegrates. This led Baran to present two alternative architectures as alternatives to the centralized architecture. Baran's original illustration is shown in Figure 4.

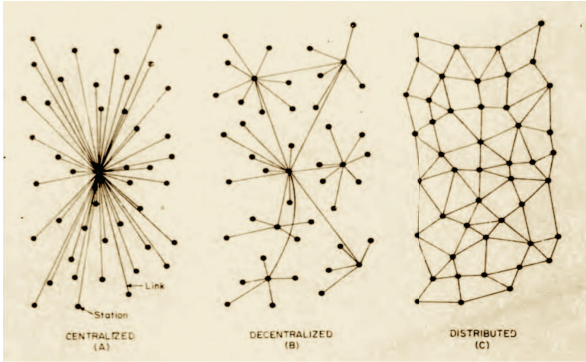

FIGURE 4. Paul Baran's three architectures for data synchronization as articulated in 1964.

The two alternatives, shown in Figure 4, are a *decentralized* architecture and a *distributed* architecture. It should be emphasized that these are not the same.

Today, the internet is a complex mixture of centralized, decentralized and distributed systems. For example, the services of the majority of internet technology companies today may be regarded as centralized. On the other hand, the vast infrastructure that moves packets of data around the world efficiently while avoiding bottlenecks via TCP/IP is better regarded as a distributed system.

The complex architecture we have today did not arrive overnight, but instead over a period of time. The first peer-to-peer message sent between two computers was in 1969 at ARPA (later DARPA), and the first email was sent in 1971, also at ARPA. TCP/IP was adopted as a standard protocol in 1983 and the first web server went live at Stanford in 1991. Finally, the World Wide Web was made freely available to the public in 1993.

The question arises: How can all these various different systems work together?

At this point we will step away from the digression into history and recall the core concept of Section 1, which is that if information is sent in a way that is trustless, it does not matter how the information is sent or what happens to it between the sender and receiver. This concept is essential for services like online banking today, and it is equally essential for the decentralized internet. In both cases, computers arrange themselves in some sort of network where they can pass messages around between each other, and the integrity of the system can be maintained by ensuring that the data that is being passed around is in the form of digital certificates, that is packets of data from which a receiver may infer some information without reliance on what happened to that information before they received it. An example of this is the way that digital certificates can be used to convey to a user that they are indeed viewing the website corresponding to the url in the address bar, assuming they trust their browser and operating system.

The fundamental difference between the existing internet and the decentralized internet is the following: Whereas the validity of a url in an address bar is certified using a chain of digital certificates that is only several certificates in length, the decentralized internet is based on the idea that useful data structures can be securely encoded using great multitudes of digital certificates.

For this reason, it will be helpful to introduce some formalism for understanding the different types of digital certificate and how they may interact with one another. To that end we will distinguish between three different types of digital certificate as follows:

Definition 2. *Suppose* (M, P) *is a digital certificate. Suppose* M *contains an identity marker* x *and suppose also that* P *is of the form: "The author of* M *was in possession of secret information associated with* x*." Then we say the digital certificate* (M, P) *is a* proof of authorship certificate.

Definition 3. *Suppose* (M, P) *is a digital certificate. Let* A *be a monetary amount. Suppose* P *is of the form: "At least* A *was expended to produce* M*." Then we say the digital certificate* (M, P) *is a* proof of work certificate.

Definition 4. *Suppose* (M, P) *is a digital certificate. Suppose* M *contains a data identifier* x *and contains or otherwise references data* D*. Suppose* P *is of the form: "The data* D *correctly corresponds to* x*." Then we say the digital certificate* (M, P) *is a* proof of authenticity certificate.

An example of a proof of authorship certificate is a message signed with RSA. Here, x is the public key and the secret information is the private key corresponding to x. We say that the user creating M has *signed* the message. The advent of the decentralized internet has popularized the term *wallet*, which is software similar to a password manager but instead of storing passwords a wallet stores private keys that a user can use for signing

messages. An example of a proof of work certificate is a string whose SHA256 hash is less than some given threshold. Such proofs of work are key to the Bitcoin protocol. An example of a proof of authenticity certificate is a file obtained using IPFS. In that case the identifier is the CID of the file and the data is the file itself. There are many other types of digital certificate, but these three provide the backbone of the decentralized internet.

Digital certificates do not exist in isolation, so let us now introduce some further formalism to characterize how digital certificates may exist contingent on one another.

Definition 5. *Suppose Bob receives a message M' and suppose also that Bob knows proposition P to be true. If Bob is able to independently verify the truth of proposition P' using no information other than what is contained in M' and the truth of P, we say that (M', P') is a* certificate contingent on P. *If (M, P) is itself a certificate, we abuse language and say that (M', P') is a* certificate contingent on (M, P). *We call (M', P') a* contingent certificate. *The certificate (M, P) may itself be contingent.*

Intuitively, this is formalizing the chains of digital certificates used to verify a url in a browser address bar. In that case the certificates in question are proof of authorship certificates. The final certificate in the chain says: "I am Charlie, and you can trust me because Bob says you can trust me. This is the public key corresponding to `bankofamerica.com`." The previous certificate in the chain, on which Charlie's certificate is contingent, says: "I am Bob, and you can trust me because Alice says you can trust me. You can trust Charlie to tell you the public key corresponding to `bankofamerica.com`." Eventually, to prevent the chain running forever, if the only certificates we are allowed are proof of authorship certificates, we have no choice but to eventually trust a root authority.

The central theoretical innovation in the Bitcoin protocol is that a chain of certificates does not need to terminate with a trusted root authority, but instead with a proof of work certificate. If Bob sends Charlie $100 in Bitcoin and Charlie wants to verify the transfer, he may inspect a certificate signed by Bob that says "I send Charlie $100 in Bitcoin."[4] To make sure Bob has $100 in Bitcoin to send, Charlie may inspect another certificate or certificates confirming that a person or people had previously sent at least $100 to Bob. Eventually, following the chain of certificates (also known as *transactions*) all the way back, the chain stops with one or more proof of work certificates. The person who generates the proof of work certificate also verifies that nobody in the chain of transactions sent their last $100 to multiple people, in other words that there was no *double spending*. This is achieved by giving the person who generated the proof of work certificate a bounty in the form of some amount of Bitcoin for doing the proof of work, but in order to claim that bounty they have to look through the complete list of transactions so far and certify that there has been no double spending. They also certify any new transactions as not resulting in double spending. The person who generated the proof of work certificate, known as a *miner*, could fail to obey these rules, but then they would lose out on the bounty.[5] Given that the miner has expended resources to generate the proof of work certificate, in practice they do not break the rules and instead obey the rules and claim the bounty.[6]

Summarizing what was just said, it is not the case that Charlie can inspect a collection of certificates and verify the statement: "Bob sent me $100." However it is the case that Charlie can verify a statement along the lines of: "Either Bob sent me $100 or, with very high probability[7], someone spent at least $1,000,000 to trick me." If Charlie wanted to increase the number $1,000,000 to, say,

[4]Of course, Bitcoin transactions are not denominated in US dollars but instead in Bitcoin itself. We refer to dollar amounts for ease of reading.

[5]Additionally, a person making a transaction also attaches a small bounty, called a *fee*, which is paid to the person who generated the proof of work as an incentive to record and certify the transaction.

[6]For an excellent introductory talk on the mechanics of Bitcoin which goes into more detail on the theoretical computer science of consensus protocols, see: `https://vimeo.com/314137501`

[7]We add the caveat "with very high probability" because the cost of generating a proof of work certificate is not deterministic, but exists on a distribution. The cost of generating a proof of work certificate is distributed in a similar way to the cost of generating a winning lottery ticket; with very high probability someone needs to buy a lot of tickets but in principle they could be lucky with just one. Even if a miner was very lucky and was able to generate a proof of work certificate for nearly nothing, the miner would still be incentivized to obey the rules of the protocol to claim the bounty. Thus if we replace the word "spent" with "forwent" we may remove the "with very high probability" caveat.

$10,000,000, Charlie could simply wait for more miners to produce some more proof of work certificates and see that they also verify the complete list of transactions as valid and containing no double spending. Every time a miner adds a new proof of work to the complete list of transactions, known as the *ledger*, we call the new proof of work and the associated new transactions a *block*. The collection of all the blocks is called the *blockchain*. Thus, to add extra confidence to the transaction, Charlie can wait for some *more blocks to be mined and added to the blockchain.*

When a block is added to the blockchain, it is important that it cannot be changed after the fact. For this reason a proof of authenticity certificate is used. Specifically, each new block incorporates a special number called a *hash* that is generated by the previous block. The hash is the data identifier in the definition of a proof of authenticity certificate and the previous block is the data. In addition to their other duties, the miners check that each block correctly refers to its previous block when they certify the blockchain as valid, and again if a miner failed to do so they would lose out on their bounty.

Sometimes it may be the case that two or more miners generate a proof of work at around the same time and before either knows about the other's block. This is resolved by the convention that the blockchain with the longest chain of blocks is the correct one. Thus, miners are incentivized to mine for blocks that add to the longest blockchain. For this reason, it can be the case that there are multiple versions of the blockchain with differing final blocks, but as soon as one of the competing blockchains develops a lead it leaves the others behind and becomes the authoritative ledger. In the case of Bitcoin, new blocks are generated on average every 10 minutes[8] while new blocks are broadcast between the miners in seconds or fractions of a second, so in practice multiple competing blocks don't arise very often.

It is important to emphasize that large parts of the Bitcoin protocol are unenforced. For example, the question arises as to how the miners obtain the transactions to add to the ledger. Remarkably, this is done with a *peer-to-peer network* operating on a *best effort basis*. That means that people who want to help the community do so by listening for transactions and broadcasting valid transactions to other people they are in touch with. The people who do this are said to be running *nodes*. The collection of transactions circulating between the different nodes is called the *mempool*. It doesn't matter that some, or even the majority, of nodes might be selfish and listen for transactions while not broadcasting them on. What matters is that anyone, for example a miner or someone running a node, can quickly check the validity of the transactions they become aware of. The worst a selfish or even malicious node could do to disrupt this is to stay silent as any fake transactions can quickly be identified as such and any node broadcasting fake transactions can be quickly shunned.

The Bitcoin nodes themselves may be regarded as a distributed system, whereas if one considers the nodes together with every device that connects to a node, it is better regarded as a decentralized system.

The mechanics of Ethereum is largely similar. The main difference is that instead of just keeping track of financial transactions and the state of the ledger, Ethereum also keeps track of computer code and the miners calculate the result of that code and keep track of the results. While this idea may sound simple, in practice it is much harder than it may sound because a malicious or careless user could enter some code that never terminates. For example, someone could ask Ethereum to execute code that set `i = 0` and then incremented `i` by `2` forever as long as `i` never equals `9`. If that happened, the code that generated new blocks would "hang" and never produce a new block.

One might wish to get around this problem by inspecting each program to be run on Ethereum to make sure it won't run forever before running it. Unfortunately, a famous result in theoretical computer science called the *halting problem* says this is impossible, so another approach is needed.

There are two main alternative approaches that have been used. The first way is to restrict the types of programs that can be executed to not allow loops that might hang. This is in fact the approach used in Bitcoin. The way that transactions are recorded in Bitcoin is somewhat more complicated than described earlier in this section. Instead of Bob signing a certificate that says "I send $100 of Bitcoin to Charlie," he signs a certificate that says something like: "This $100 of Bitcoin can now be

[8]This is ensured by programmatically adjusting the difficulty of the proof of work.

sent by anyone able to give inputs to the following code that returns `true`." To make sure only Charlie is able to send the Bitcoin on, Bob uses Charlie's public identifier in such a way that only someone knowing the associated secret information can make the code return `true` with a proof of authorship certificate. Here, the code is written in a language called *Bitcoin Script* and it has the property that it does not allow loops that might run forever. While preventing programs from hanging, this has the downside that Bitcoin Script is not suitable for developing real applications beyond those concerned with moving Bitcoin between users and other very simple toy applications. Because Bitcoin Script does not allow certain types of computer programs, we say that Bitcoin Script is not *Turing complete*.

The approach Ethereum takes is different. Instead of using a restricted programming language like Bitcoin Script, Ethereum allows users to give instructions using a Turing complete programing language, effectively allowing a user to write any program. To prevent programs from running forever, a user must state in advance how many computer operations will be needed and pay for those computations. If they underestimate the number of computations that will be needed their program will not be executed. Similar to the way Bitcoin transactions include a transaction fee to incentivize miners to include the transaction, people executing code with Ethereum include a similar fee called a *gas fee* to incentivize the miners to include their instructions in the next block. The net result is that, instead of relaying certificates containing financial transactions, the Ethereum protocol enables Bob to verify something along the lines of: "Either Alice executed the following function resulting in the following state in an abstract computing environment or, with very high probability, someone spent at least $1,000,000 to trick me."

In recent years, there has been a movement to an alternative to proof of work called "proof of stake." The distinction between proof of work and proof of stake is not important for our purposes. What is important is that proof of stake facilitates a recipient to verify a statement to the same effect as with proof of work. It is not theoretically impossible to change a transaction on Bitcoin or Ethereum after the fact, however it is cost prohibitive to do so.

So far we have considered how one may simulate a computing environment with no central server. Using a system such as Ethereum one could in principle develop any application. However the key question remains: Is this system fast, scalable and cost effective?

Unfortunately the answer to the question is: No. There are two main reasons for this. First, with Ethereum every miner not only stores the computer instructions, but also evaluates them. That means every line of code written by a developer making an application with Ethereum must be evaluated as many times as there are miners. Second, even for an application that is only of interest to two people, the entire network has to participate in checking the computation is valid and storing the relevant information.

Recent years have seen tremendous progress regarding both of these problems. Regarding the first problem, *Bitcoin Computer*[9] is an approach to computation using a blockchain which only stores the instructions on the blockchain, but leaves the computation to be carried out by interested participants. That results in a reduction in the cost of computation of many orders of magnitude as compared to Ethereum, but still leaves the question of storage.

Regarding that second issue, in 2011 Shapiro et. al. [6] presented the concept of a Conflict-Free Replicated Data Type (CRDT), which is a method of maintaining consensus across a network of computers. CRDTs lack the long-term security of a blockchain like Bitcoin or Ethereum, but present a method for computers to connect to each other and maintain consensus without relying on a central server and at close to zero cost.

We will conclude this section with the question of the scalability of the digital certificates themselves, rather than the overall system. For technical reasons, proof of authorship digital certificates cannot be too big. For example, a message signed using RSA-2048 is limited to just 245 ASCII characters. While one could extend that limit somewhat by concatenating multiple messages, for any larger data type, an image or video file for example, it is not practical to sign the message itself with a proof of authorship certificate.

Instead, what can be done is that the image or video file can be authenticated with a proof of authenticity certificate. The resulting data identifier,

[9]See: `https://www.bitcoincomputer.io`

often called the *hash* of the file, is short enough to be signed with a proof of authorship certificate. Thus, when building an application using technology such as Ethereum, one would typically not store large files themselves, but instead only the hashes of the files. This is no great loss, since a user is able to verify the authenticity of the file using the proof of authenticity certificate, provided they are able to obtain the file itself.

An example of a system that enables the sharing and storage of files is IPFS. In that system, files are identified with the hash of the file itself, supported by a distributed peer-to-peer network to store and propagate the files. Similar to how transaction fees and gas are used, respectively, in Bitcoin and Ethereum, Filecoin adds an intricate incentive layer to IPFS to encourage file storage. Users wishing to avoid using the incentive layer can simply pay companies like Pinata in fiat currency to store their files on IPFS and make them available to the network. Because of the use of proof of authenticity certificates, someone receiving a file could be completely indifferent as to where the file came from, how its storage was paid for, or the protocol by which it was communicated.

There is a great deal more to discuss regarding the decentralized internet, but the broad themes have been laid out. In the manner described above, a sophisticated user can access applications that are trustless contingent on minimal assumptions about the integrity of the cryptography underlying the individual certificates and the incentive-following behavior of the miners operating the various decentralized systems. Moreover, recent developments mean that meaningful decentralized applications can now be built that are computationally intensive, reliant on large amounts of data, fast, with strong guarantees as to the cryptographic soundness of the underlying data structures, and all at reasonable cost.

However, all this progress is worth very little without a way of conveying to a user what is and is not to be trusted. The question of communicating trust is the subject of the following section.

4. Enso: An overview

To fully describe Enso will take some time. It is part web browser, part wallet and part node. It also critically relies on online services to ensure a fast and seamless user experience. However, in this section we will put aside technical details and focus on what a user actually sees, and how they may understand and interact with what they're seeing.

Enso is built around the idea that a user should be able to understand what is true through their user interface. To facilitate this, it is necessary for a user to understand that there are certain types of experience they may encounter that a malicious developer could use to lie to them, while there are other types of experience that a malicious developer would not be able use as a tool for dishonest communication.

The simplest first step to achieving this is to separate the screen into distinct areas which may or may not be available to a developer to manipulate. In a regular browser this is achieved by rendering the web app in a rectangle of screen, while reserving the top part of the window for communications from the browser itself, including the address bar and padlock. This is illustrated in Figure 5.

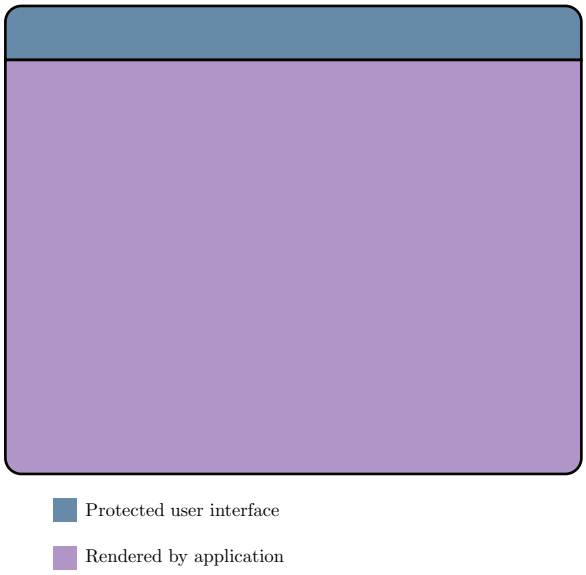

Figure 5. Protected versus non-protected user interface areas in a normal browser.

While a thin area of the screen at the top of a browser window is big enough for an address bar, including padlock, and a few other user interface items, it is not big enough to show anything involving larger portions of text.

One way to overcome this, as used by browser extension wallets such as MetaMask, is to break the line that separates the protected part of the

browser window from the part rendered by the application with an overlay box that itself contains protected user interface. While an application developer can send messages to that part of the interface, they may not draw on it directly. This is illustrated in Figure 6.

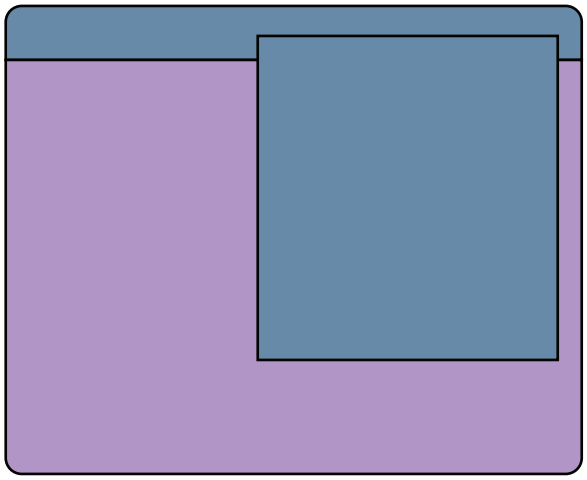

FIGURE 6. Extending the protected part of user interface by "breaking the line."

Let us consider a simple example. Suppose that Bob (full name Bob Smith) wants to make a public social media post that says: "Today I ate breakfast. Everyone should eat breakfast." When Alice sees Bob's post, she wants to know that Bob really made the post. Let us examine how this can be achieved with what we have introduced so far.

Using RSA or something equivalent, Bob can sign a proof of authorship certificate that says: "Today I ate breakfast. Everyone should eat breakfast." Bob then disseminates the certificate in a way that Alice can receive. It doesn't matter how this is done. Bob could publish his post via an HTTP server, or store it on a blockchain, or use a peer-to-peer network like IPFS, or send it to Alice directly. However Alice receives the certificate, upon receipt she can assure herself by inspecting the signature that someone with a certain public key signed the message. Therefore, it would be possible to develop a user experience that looks something like Figure 7.

While Alice may feel reassured that the post that claims to be by Bob has been signed by someone, unless she knows that the displayed public

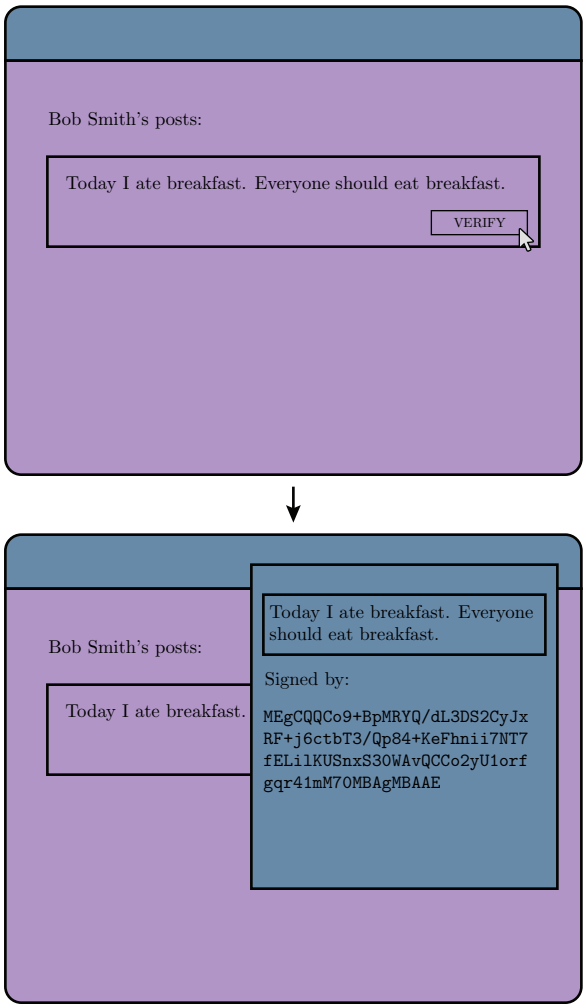

FIGURE 7. The limitations of using public keys to identify people.

key corresponds to Bob Smith, she doesn't have any information at all about whether Bob indeed authored the post.

We have discussed problems very similar to this in Section 1. The problem of matching Bob Smith to a public key is structurally the same as matching a url like `bankofamerica.com` to a public key. If that can be done in a way that Alice trusts, she would be able to know that Bob indeed authored the post.

There have been a number of attempts to match identities to public keys that do not rely on trusted authorities. One approach is to use features of a person's real identity, their name, date of birth and so forth, as a seed to generate the public/private key pairs commonly used for proof of authorship

certificates. This is called *Identity-Based Encryption* (IBE). See [1], for example. Another approach is to use *webs of trust*. These are peer-to-peer networks in which the nodes share with each other the public keys of identities they know. One of the main implementations of this idea was *Pretty Good Privacy* or *PGP*. See [5], for example.

Neither approach has been able to displace trusted authorities. The first approach, while intuitively appealing, encounters the problem that as identities are public, anyone who knows how to convert a real identity into a public/private key pair would be able to sign anyone's messages. This is known as the *key escrow problem*. Keita et. al have recently proposed a solution to the key escrow problem [3], but their solution is highly technical and also relies on the presence of a trusted authority to certify identities, obviating the intention of not relying on a trusted authority.

Regarding the second approach, webs of trust run into the problem that they rely on users to take the time to build up an address book of known public keys and their corresponding real identities. Figure 8 shows a party where participants are exchanging public keys with one another[10], which perhaps sheds some light on why webs of trust have never gained widespread adoption with ordinary users of the internet[11].

FIGURE 8. One approach to building a list of known public keys.

In any event, it would be fair to say that finding a robust, trustworthy alternative to the trusted certificate authority model is an area of ongoing research. It is possible that in the future there may emerge a reliable way of knowing that a certain public key corresponds to a certain identity without making use of a trusted authority, but no such method exists at this time.

Thus, for now we are forced to accept the need for certificate authorities in order to build applications that use real identities and we will assume that a trusted authority has certified the connection between Bob's public key and Bob Smith's real identity. For the sake of our presentation, we will call this trusted authority Web3 Identity Services, Inc. In this way, we may create an interface that looks something like that shown in Figure 9.

Of course, Bob Smith is likely not to be the only person by that name, so it make sense to disambiguate him from the other people called Bob Smith. To achieve this, note that Bob Smith has many other characteristics, such as his age, where he went to school, places of employment, geographic location, and so forth. This information, similar to what one might find on a LinkedIn profile page, can all be certified by appropriate authorities, and that can in turn be presented on the screen in the overlay area of the user interface.

The drawback of this approach is that if we fill the overlay box with too much information it either has to extend far down the page so that the lower portions can only be reached by scrolling the main window, or alternatively the overlay panel itself has to scroll. Both these solutions are unsatisfactory as they constrain a page that should take up the whole window in the small overlay box.

Thankfully, there is an alternative approach. We have said previously that the screen is divided into a trusted area at the top and an area that an app developer can use at the bottom. While the bottom area does not have control over the top area, the reverse is not the case as the interface can control everything on the screen. Provided that there is an indication in the top area of screen that the bottom area of screen should be trusted,

[10]Image courtesy of Wikipedia.

[11]For an informative post on the limitations of the web of trust model, see: `https://medium.com/@bblfish/what-are-the-failings-of-pgp-web-of-trust-958e1f62e5b7`.

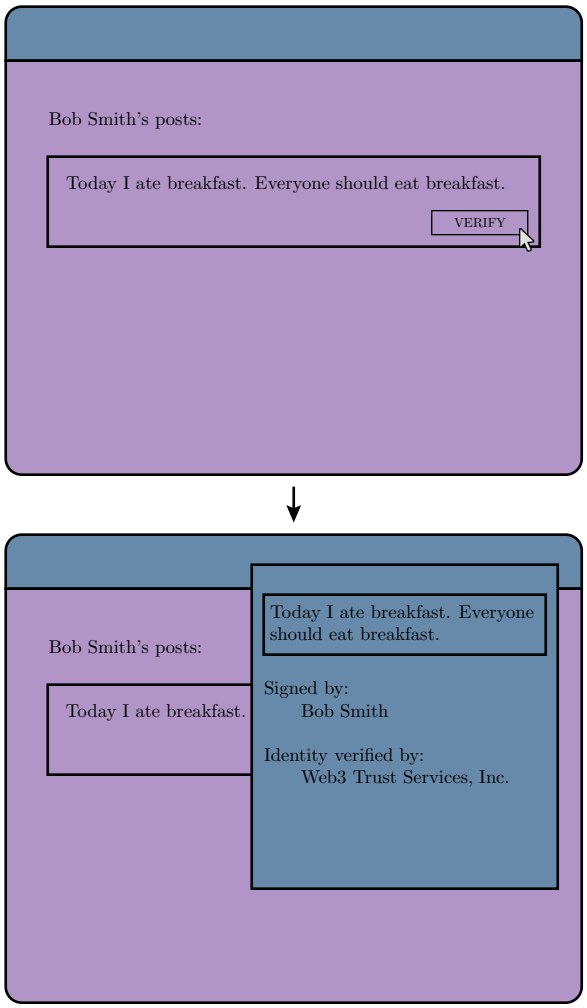

FIGURE 9. Using a certificate authority to connect real identity with a public key.

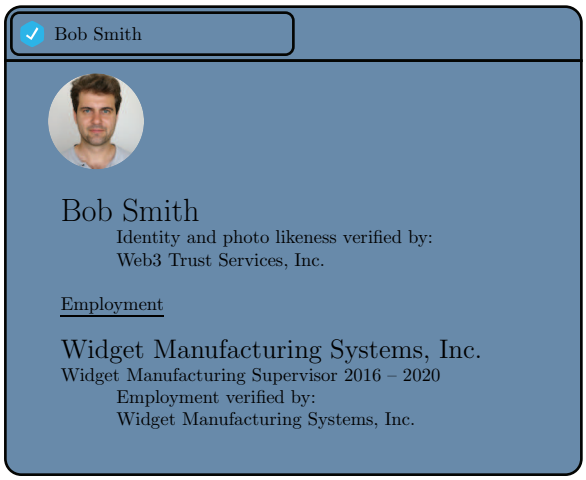

FIGURE 10. Provided the user knows to check the trusted area of the screen, the entire window may be used to display trusted content.

and the user understands that only the top area of screen is able to make this declaration, we may use the entire screen to convey the information we desire about Bob Smith. This is illustrated in Figure 10.

The question arises now, if Bob's employment is verified by his employer, how is the employer verified? This turns out to be a rather thorny question; an institution like a company does not act for itself but instead has authorized people, executive officers and directors and such, who act on its behalf. Those authorized representatives in turn need to be verified, and their association with the institution also needs to be verified. What if Bob was not a regular employee at Widget Manufacturing Systems, but instead its owner and CEO? Could he then certify himself? These questions require careful thought, however questions of this sort are not new, meaning there is a rich literature on how to approach the connection of real and digital identities using digital certificates. See [4], for example. In very brief summary: Bob is not allowed to certify himself. Instead we insist that everything that is being vouched for must eventually make its way back to a root authority via a chain of delegated authority. If the chain runs around on itself, that is not valid.

Thus, we will assume that it is possible to view a page for a person or institution and see reliable information, that is information that has been certified as true by trusted or legitimate authorities, and the chain of certification is visible to a user if they wish to scrutinize it.

Let us then return to Bob's public post, and Alice's desire to see that he really did make the post that she is being shown. One approach is that when Alice clicks VERIFY in the top image of Figure 9 she could be taken to a section of Bob's profile (Figure 10) showing that post. There is certainly nothing stopping the profile page of Figure 10 from containing Bob's properly signed public posts, however the issue arises that an application developer who wishes to display Bob's public posts may not want to take the user to another page every time the user wants to verify the integrity

of a post. This would especially be the case if a page contained many signed certificates, all of which could be scrutinized by the user, cluttering up the page with numerous VERIFY buttons.

So, we want to enable a developer to show trusted information that is generated by a potentially large number of signed certificates. However we also want to prevent a malicious developer from abusing whatever freedom they have by lying to the user. The key question is: Should an app developer have complete control, on a pixel by pixel basis, of the lower, untrusted, part of the screen?

It is reasonable to say the answer should be 'no'. An example of a system that works in this way is a bot on Slack. There, a developer has control over posts and messages and such, but cannot draw arbitrary content on top of the user interface.

Enso takes the opposite view. App developers can draw arbitrarily on the rectangle of screen reserved for them. In fact, if a user enters a normal url like `https://google.com` then Enso works for the most part like a normal browser. The top, trusted part of the user interface shows the url and padlock (assuming a valid SSL certificate), while the bottom part shows content written in HTML, CSS, and Javascript. However, whereas a normal browser gives an implicit stamp of approval to any website that has a valid SSL certificate, Enso gives no such reassurance. When a user visits a normal website in the normal way with Enso, they are warned that there are no guarantees about the truthfulness of the content they are seeing. This is illustrated in Figure 11.

FIGURE 11. A secure connection, but not necessarily a trustworthy source.

We now reach the following question: If application developers can draw arbitrary content on the untrusted portion of the screen, what is to stop them lying to the user? One approach that has been used, for example by the Apple app store, is to audit every app available for users. This approach has many advantages for users who want a very safe experience, but it dramatically slows down the development and deployment process, and allows for rent seeking behavior if there is a single company acting as sole gatekeeper.

Enso approaches the question of conveying trusted content as follows. First, it is the case that an app can be audited and if it has been audited then the user can be informed of this. Second, the certification of satisfactory audit status is delegated in a similar way to the way SSL certificates are delegated. In this way, a developer can produce an interface that looks something like that shown in Figure 12.

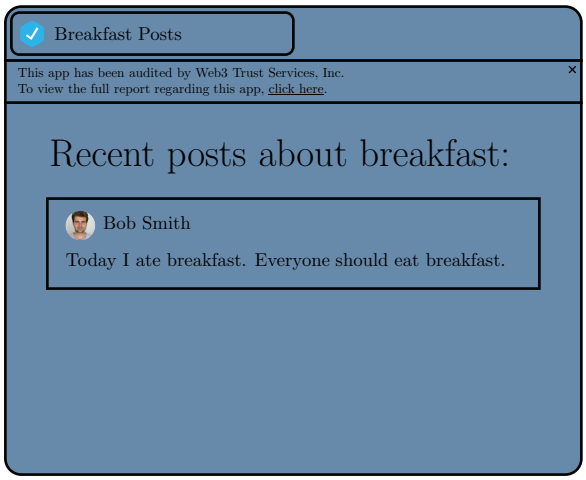

FIGURE 12. If an app has been audited and the user trusts whoever did the audit, then the whole screen can be used to convey trusted information.

What has been introduced so far would be a very satisfactory basis for an interface for web3 applications. However, it would necessarily be a somewhat slow process to release an app if every time the developer wanted to make an update they had to resubmit their application for audit. Ideally, a

developer would be able to display trusted content in their application, while at the same time not having to submit their application for audit.

Enso achieves this by enabling apps to display trusted content over the untrusted part of the screen without the need to break the line between the trusted and untrusted part of the screen. Specifically, when a user starts using Enso, they enter some secret information, for example a hand-drawn circle, and that is used to designate a trusted part of the screen. This is illustrated in Figure 13.

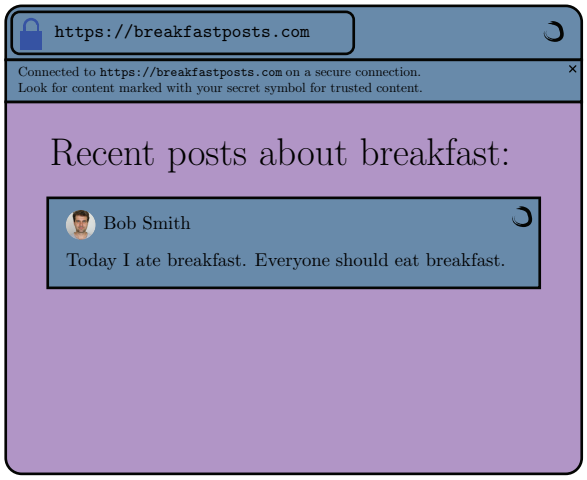

FIGURE 13. Trusted content may appear in the middle of the page provided it is marked with secret information a developer does not have access to.

A trusted box is simply an app that has been audited drawn in a window that floats over the untrusted part of the interface. Note that it is *not* an iFrame for the simple reason that an application developer could draw over the top of an iFrame and thereby trick the user. Instead the trusted box is a completely new browser view that happens to sit on top of the main application for the page. It is an audited application that happens to be in a smaller window positioned over the top of another application.

So far we have mainly been concerned with how a user may understand that they are being shown accurate information. To recap: Sometimes they

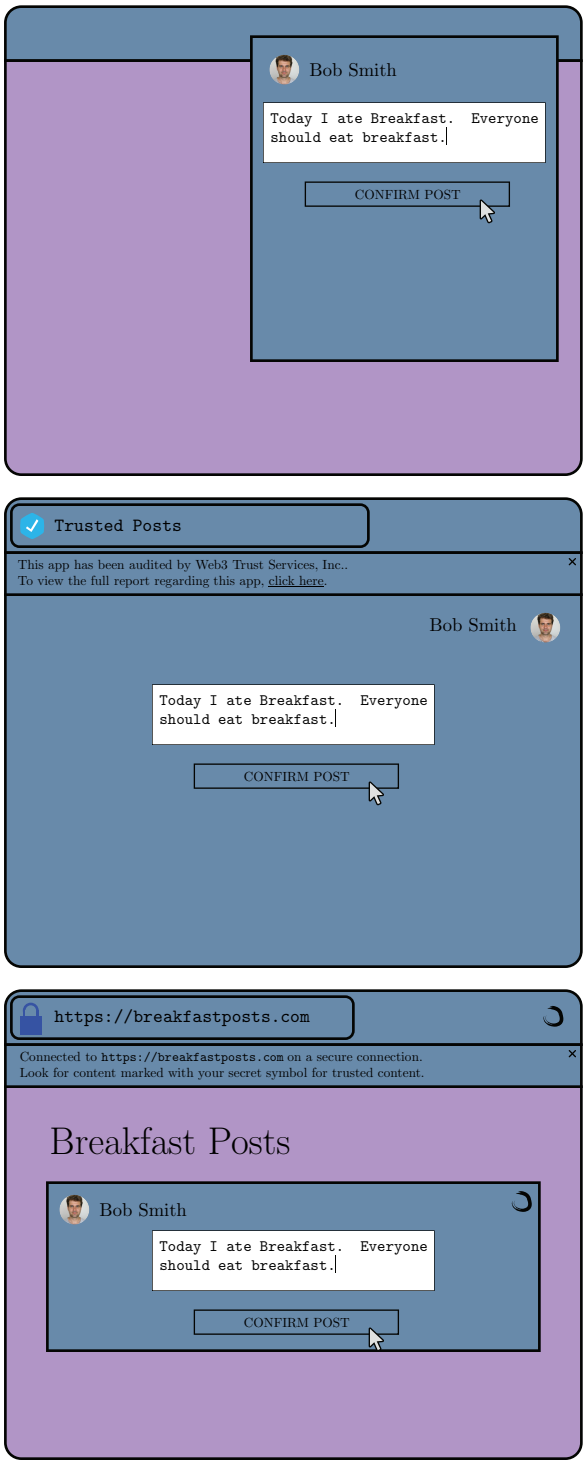

FIGURE 14. Three alternative ways to confirm a user action in a trusted manner.

may see an overlay box that breaks the line separating the trusted and untrusted parts of the window. Sometimes they may see that an application has been audited in which case they can trust everything in the window contingent on trusting whoever did the audit. Sometimes there may be an audited application within another application, marked by secret information. Finally, they might see a normal website in which case they are warned that while the connection may be secure, there are no guarantees about the trustworthiness of what they are seeing.

Let us now turn to the question of how Bob creates his post and signs it with his private key. Similar to with the display of information, it is important that Bob knows which part of the user interface is to be trusted and which part isn't. This is achieved in the exact same way as with information display, as shown in Figure 14. Crucially, with audited applications, either standalone or inside an untrusted application, part of the audit process is the verification that accurate human readable descriptions have been provided for what the application is doing when a user takes a certain action.

Indeed, this principle of showing which parts of the interface can be trusted and which parts cannot necessarily be trusted extends to everything a user does with Enso. These actions include sharing data with specific other users, publicly broadcasting data, sending money or cryptocurrency to another user or organization, making purchases of physical or digital products and updating information stored either locally or remotely.

5. DNS with Enso

Enso is in many ways similar to a web browser. In fact, if one defines a web browser as software that renders applications from the internet, written in HTML, CSS and Javascript, in a rectangle on the screen, then Enso *is* a web browser. However, if one defines a web browser as software that makes an HTTP request to a server and displays the result on the screen, then Enso is much more than a web browser.

When a url is entered into the address bar of a normal browser, the domain name is resolved to an IP address using DNS, and then that IP address is sent an HTTP request. In other words, a web browser cares about *where* the request is responded to from.

The actual mechanics of DNS lookups is quite involved and not important for the present discussion. When a user enters a normal domain name such as `https://breakfastposts.com` in the address bar in Enso, the DNS lookup is exactly as it is in a normal browser.

It is worth contrasting the approach of specifying an application with a domain name with the approach taken by a system such as IPFS. Using a suitable browser extension[12], a user can enter an address such as `ipfs://QmbWqxBEKC3P8tqsKc98xmWNzrzDtRLMiMPL8wBuTGsMnR` to retrieve a web application. Here, the text following `ipfs://` is the data identifier for a proof of authenticity certificate. In other words, specifying a webpage using a proof of authenticity certificate such as an IPFS CID cares about *what* the resulting application code is. This use of proof of authenticity data identifiers for identifying content has its benefits, but at the cost of the identifier no longer being meaningful or memorable to humans.

This tension between human readable domain names such as `https://breakfastposts.com` and non-human readable identifiers such as IPFS CIDs has been formalized with the concept of *Zooko's triangle* [8]. In 2001, Zooko Wilcox-O'Hearn described a trilemma between a naming system being decentralized, secure and human-meaningful. Wilcox-O'Hearn conjectured that a naming system can have at most two of the three properties. Thus, today's regular domain name system is an example of a system that is human-meaningful and secure, but not decentralized, while IPFS CID's are decentralized and secure, but not human-meaningful. Figure 15 illustrates Zooko's triangle.

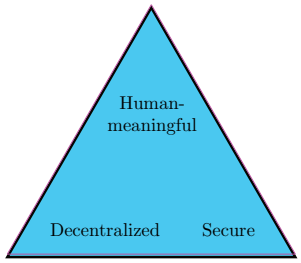

Figure 15. Zooko's triangle presents some of the tradeoffs to be considered in naming schemes.

[12]See: `https://github.com/ipfs/ipfs-companion`

In recent years, there have been proposals for alternative naming systems utilizing blockchain technology that possess all three of the properties proposed by Wilcox-O'Hearn. An example of such a system is Urbit, which uses names that are somewhat readable like `~fipbud-binzod` or `~tarwed-mostun`.

While there is a certain mathematical elegance to the Urbit naming scheme, it has yet to be proven that solutions to Zooko's trilemma of this sort will be attractive to the general public.

Taking a different approach, Nick Szabo has written persuasively [7] that the crucial feature of a namespace is widespread adoption, articulating a model wherein competing namespaces emerge over a period of time according to how well they serve their community.

Following Szabo, Enso embraces the view that namespaces should emerge over a period of time, remaining agnostic to how users specify an application. If a user wishes to specify an application by entering an address that starts with `https://` they may. Similarly, if they wish to use IPFS to load a page by entering an address that starts with `ipfs://` they can do that too. This allows for ordinary users of the internet to continue to use the internet much like they are used to. Additionally, today's users of the various web3 technologies may use applications built using these technologies. Most important of all, we hold open the possibility of new technologies and protocols emerging in the future with minimal friction to the end user. Over time, technologies and associated namespaces that are effective and popular with users will be widely adopted, while those that are less so will not be.

In other words, Enso is not just agnostic as to *where* information comes from, it is also agnostic as to *how* (i.e. according to which protocol) that information is delivered.

Instead, Enso scrutinizes application code after it reaches the client. In order for an application to be rendered with Enso and be vouched for as a trusted application, it must *somehow* deliver to the client the following pieces of data:

- application code;
- assets, such as image files, required by the application code;
- metadata including the application name;
- data identifier as used in a proof of authenticity certificate corresponding to the application code;
- proof of authorship certificate in which the author of the application signs the application code data-identifier;
- real identity of the author of the application;
- signed certificate from the identity provider connecting the real identity of the author of the application to their public key;
- if the URL is human-readable, then proof that the author of the code has control of the URL entered in the address bar[13]; and
- signed certificate from the audit provider that the code is sound.

Just as public keys associated with SSL certificates are not especially interesting to the average user of today's internet, much of the above information is not especially interesting to an end user of Enso. The two main pieces of information that an end user cares about are the name of the application and that the application is trustworthy. Additionally, they may be interested in the author of the application, and a means of creating a sharable link to send other people.

Focusing on what is interesting to the end user, and hiding technicalities necessary to ensure the information shown to the user is correct, we may create an interface that looks something like that shown in Figure 16.

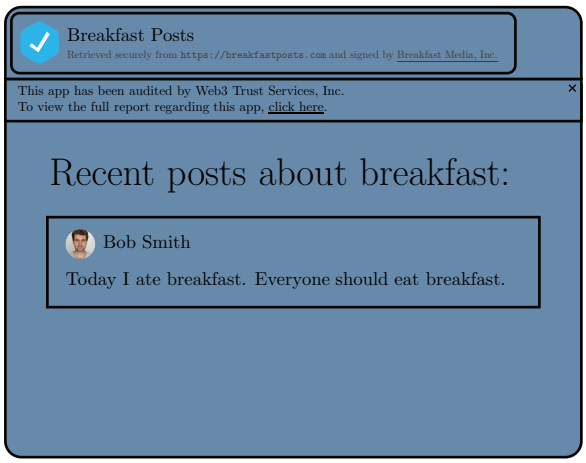

FIGURE 16. A way to enable a user to identify a trusted application

[13]For addresses that make use of a proof of authenticity data identifier, such as an IPFS CID, there is no need for this.

Beyond hiding public keys and other technical details, the question arises as to whether there are other things that can be hidden from the end user to reduce clutter. Specifically, in today's internet it is very important that a user can see the url and padlock so they know they're connected to who they think they're connected to. For example, if a user visits `https://bankofamerica.com`, the only assurance they have that they're really connected to Bank of America is the url and padlock.

In our case, we have solved the issue of reassuring the user they really are connected to the person or institution they think they're connected to using public key infrastructure to certify identities directly, rather than by using a url as a proxy. Therefore, the purpose of the url is reduced to creating a link that can be shared with other users. Thus, we can simpify the interface further by removing the url altogether, as shown in Figure 17.

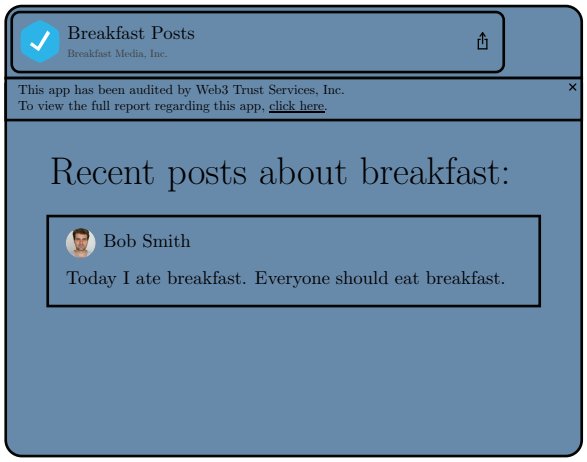

FIGURE 17. Removing the url altogether for trusted applications.

Now, when we look at where we have arrived in Figure 17, the cleanness of the interface is rather spoiled by the message explaining that the app has been audited. While this is important, it seem redundant to keep this in view for the entirety of the time that a user is using the application, or indeed whenever they return to the application. It is rather equivalent to telling a user that a website has a valid SSL certificate every time they visit that website. It would be better to hide the details of the audit message behind a simple icon, similar to how details about SSL certificates are hidden behind a padlock. We can achieve this by hiding the audit message behind a symbol in the top, trusted part of the interface.

Before turning to how to efficiently communicate the audit message to the user, is worth recalling that there are several pieces of information that a user is being reassured of. Specifically, when a user makes use of a trusted application with Enso they are being told that:

- The application author matches what is displayed on the screen.
- The application code has been audited and contains no malicious or dishonest code.
- The name of the application is an honest description of what the application is.
- The application has been retrieved in a cryptographically secure way, for example using HTTPS with a valid SSL certificate.

It would be possible to display a separate icon for each of these. However a regular user of the internet would be far better served with a simple indication of safety indicated with a single symbol that summarizes all of these things, and we will simply redefine the checkmark introduced previously to indicate all the reassurances just listed. Note that the entity that is being vouched for is the application itself, not the author of the application, so we move the checkmark over a little, next to the name of the application. This leaves room for an icon to represent the application, chosen by the author of the application, similar to a favicon. All this is indicated in Figure 18.

FIGURE 18. Bundling all the technicalities behind a clear and easy to understand checkmark.

When a user clicks on the checkmark next to the application name, they are shown an interface that exposes the various features of the application that have been vouched for, similar to what happens when a user clicks the padlock on a normal browser.

6. Developing apps with Enso

A major hurdle to the widespread adoption of the decentralized internet is the divergence away from the type of programming most developers are familiar with. Moreover, while developing for decentralized systems is hard, developing a normal web application has become easier and easier.

To illustrate this, it is noteworthy that today a developer only needs to know one language, namely Javascript, to develop a full-stack web application. This can be accomplished using Node.js on the backend, and an integrated suite of associated backend functionality such as that exposed by Firebase makes even challenging aspects very easy.

Enso's approach to development is to enable ordinary developers to access the various sophisticated functionalities of the decentralized internet by injecting into global scope an object called `enso`.

Like Firebase, the `enso` object has a variety of powerful methods attached to it. Unlike Firebase, each one of these methods not only updates the state of the sandboxed runtime environment in the browser and the rectangle of screen being rendered, but also exposes user interface motifs that extend beyond the rectangle rendered by the application in a controlled and predictable way. Also unlike Firebase, the functionality offered by `enso` enables file system access and features only achievable by running a server on the client computer.

Let us take for example how an IPFS file might be added in this system. A user clicks upload on a button shown on the screen which says "Add to IPFS" and the function `enso.ipfs.addfile` is called. This opens a dialogue box showing the user their file system, much like a normal file upload box, however the confirmation button says "Add to IPFS". The function `enso.ipfs.addfile` returns a promise, which is resolved when the file is added or the action is canceled by the user. In code these instructions can be specified by a developer in a manner similar to this:

```
enso.ipfs.addfile(
  \\ parameters
).then(result =>
  \\ handle result
).catch(error =>
  \\ handle error
)
```

Similarly, suppose a developer wishes to enable a user to send Bitcoin to another user. This can be achieved with code along the lines of this:

```
enso.bitcoin.send(
  \\ parameters
).then(result =>
  \\ handle result
).catch(error =>
  \\ handle error
)
```

As new technologies emerge, `enso` will develop new functionality, meaning that Enso is both backward compatible with the existing internet and forward compatible with emergent technologies.

Some functionality will be available to apps irrespective of whether they hold an audit certificate. Other functionality, notably that associated with the use of a user's private keys, will require a valid audit certificate.

An ordinary user of Enso will not be expected to enable all the web3 functionality exposed with `enso` when they start using Enso. Instead, whenever they encounter a technology they have not yet used, they will go through the necessary onboarding process, and this onboarding process is abstracted away from regular application developers.

7. Payment model

A key question that arises when considering how applications are deployed is the question of who pays for the infrastructure necessary to keep those applications working.

On today's internet, for the most part it is application developers who pay for hosting. To cover this cost, either directly or indirectly they pass it on to their users. Sometimes this is explicit, and other times it is less explicit, for example when the application developer has a business model based around advertising.

When an application developer launches an application for use with Enso, all they have to do is post it in a location where Enso can retrieve the

necessary data to interpret the application. If they do this in a way that is persistent, for example by storing their application on the Bitcoin blockchain, the application will be available effectively forever, provided there is a mechanism for Enso to access the data stored on the blockchain.

The ability for Enso to access, and access quickly, information stored on a blockchain is no small detail. One approach would be for each instance of Enso to store locally the whole of every blockchain of interest, in other words for Enso to act as a node for each blockchain of interest. This is not a practical approach. The Bitcoin blockchain alone is nearly half a terabyte at the time of writing while the Ethereum blockchain is more than 800 gigabytes.

What is needed is a service that scans the various blockchains, or related technologies such as IPFS, for digital certificates of interest, and makes them available via a fast, low latency service. Such services exist today, but in general are more tailored for financial trading data than general application data.

With Enso, the end user pays for this low latency access to blockchain data. Of course, many users may resist paying, but the benefit is that application developers can launch applications that scale infinitely and at zero cost to the developer.

This model of separating application development and deployment from database administration is illustrated in Figure 19.

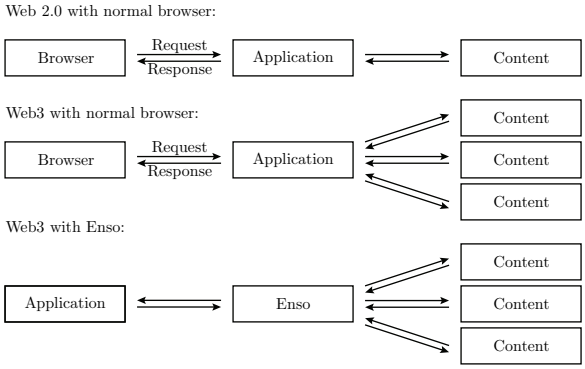

FIGURE 19. How requests are handled in a normal browser versus Enso.

8. PRIVACY

Like other aspects of application development, Enso approaches the question of privacy with the goal of providing the user with transparency and control regarding what applications are doing. For some information, for example the information a user shares on a public profile or publishes via public social media posts, the issue of privacy does not arise.

For other types of information, a user may reasonably want to maintain tight control over how that information is shared. We have already mentioned in Section 6 one example of this, namely how Enso approaches access to a user's private keys. Enso approaches the general question of private data in a similar way. Specifically, just as access to private keys is mediated through calls to methods exposed by the `enso` object, which in turn are subject to audit, private data stored in Enso is accessed via the `enso` object and, depending on the specific use case, subject to audit.

Consider, by way of example, a company called Shoes, Inc. which sells shoes. Suppose a user wishes to purchase a pair of shoes for $100. To achieve this it is necessary to send the company the information it needs to fulfill the order, as well as $100. This can be communicated via an interface as shown in Figure 20.

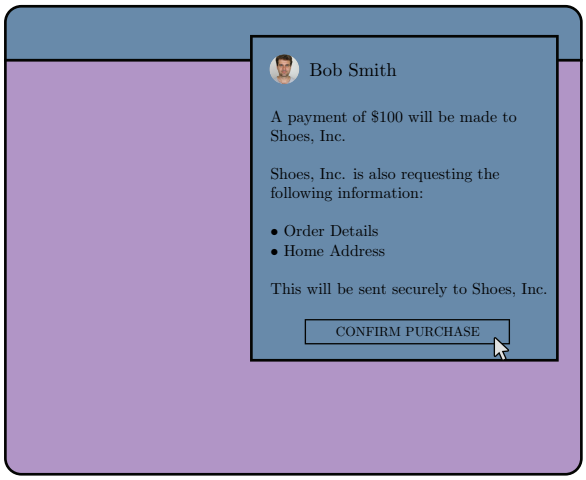

FIGURE 20. Confirming the sharing of private information with Enso.

The specific manner in which the personal information is sent to the application owner is not important to the user, so it is not elaborated on in the interface. One way to achieve this would be for the application developer to expose an HTTP endpoint and for the data to be sent to an API at that

endpoint. The connection between the entity being sent the data and the endpoint is established using public key cryptography as previously discussed.

References

[1] Joonsang Baek, Jan Newmarch, Reihaneh Safavi-Naini, and Willy Susilo. A survey of identity-based cryptography. In *Proc. of Australian Unix Users Group Annual Conference*, pages 95–102, 2004.

[2] Paul Baran. On distributed communications networks. *IEEE transactions on Communications Systems*, 12(1):1–9, 1964.

[3] Keita Emura, Shuichi Katsumata, and Yohei Watanabe. Identity-based encryption with security against the KGC: A formal model and its instantiations. *Theoretical Computer Science*, 900:97–119, 2022.

[4] Joon S Park and Ravi Sandhu. Binding identities and attributes using digitally signed certificates. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 120–127. IEEE, 2000.

[5] K Ryabitsev. PGP web of trust: Core concepts behind trusted communication, 2020.

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[7] Nick Szabo. Secure property titles with owner authority, 1998.

[8] Zooko Wilcox-O'Hearn. Names: Distributed, secure, human-readable: Choose two. 2001.