Ursa Library Motivation

Below are the reasons why the Ursa lib was created; explained as plainly as possible.

Point #1: All in One Place

Currently, each of our blockchain platforms directly use cryptography libraries written by other developers and unfortunately each project has chosen to use a different implementation. At first this seems reasonable since each blockchain was designed to serve a different niche. Indy uses a crypto library that has the features they need for the complex digital signature and proofs stuff they are doing. Iroha uses a crypto library that is designed to work on embedded/mobile devices. Fabric uses one that is time tested and trusted. Sawtooth uses time tested crypto but also has secure enclave support for PoET. Each blockchain had different requirements from the others that influenced their choice in crypto libraries and that was fine until now. Figure 1 below illustrates the current situation.

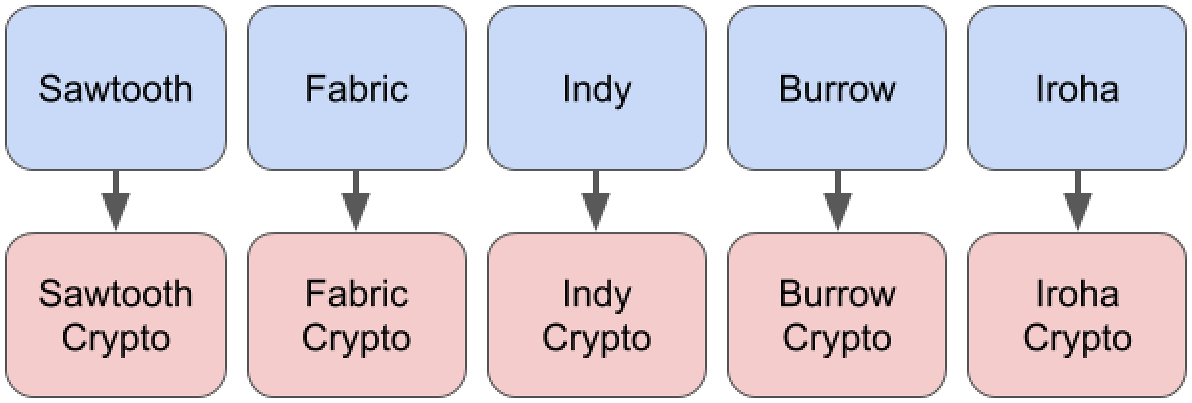

Figure 1 — Currently each blockchain uses their own specific crypto implementation.

As certain features in one blockchain gain popularity, the other blockchains want to adopt those features to stay competitive. The problem with the current situation reveals itself when the Fabric team wants to support the same signature/proof schemes as Indy or the Sawtooth team wants to support embedded /mobile systems or the Iroha team wants to use a secure enclave. All of our teams want to cherry pick crypto features from the other projects which creates duplicate code spread throughout Hyperledger as illustrated in Figure 2. It is a maintenance nightmare because fixing security issues found in one crypto implementation requires developers to go through all of the projects and make sure each copy is updated.

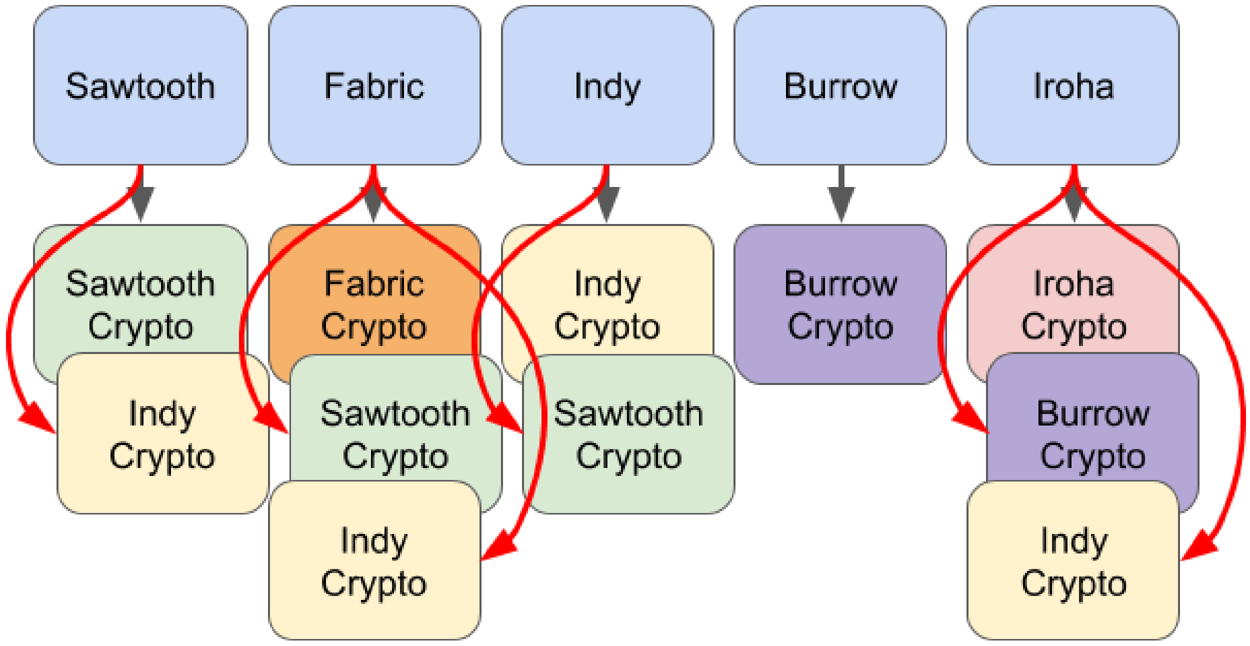

Figure 2 — Sharing through duplication is not maintainable.

The answer to this problem is to gather up all of the crypto implementations and put them into a single meta library that everybody uses. Ursa is that meta library. It ensures that there is only one copy of each crypto implementation in all of Hyperledger. There is only one copy of the complex signature/proof code from Indy. There is only one copy of the secure enclave code from Sawtooth. There is only one copy of the embedded crypto from Iroha, etc. This allows the security team to always know where the crypto code is—which is important for regulatory compliance—and what versions of crypto code we use—which is important for keeping up-to-date on security fixes. It also means that our blockchains can, in theory, easily cherry picking of crypto features from the different implementations. But in practice that is a much harder problem and is described in detail in Point #2. Figure 3 shows how Ursa will save our blockchain developers from having to duplicate the crypto implementation.

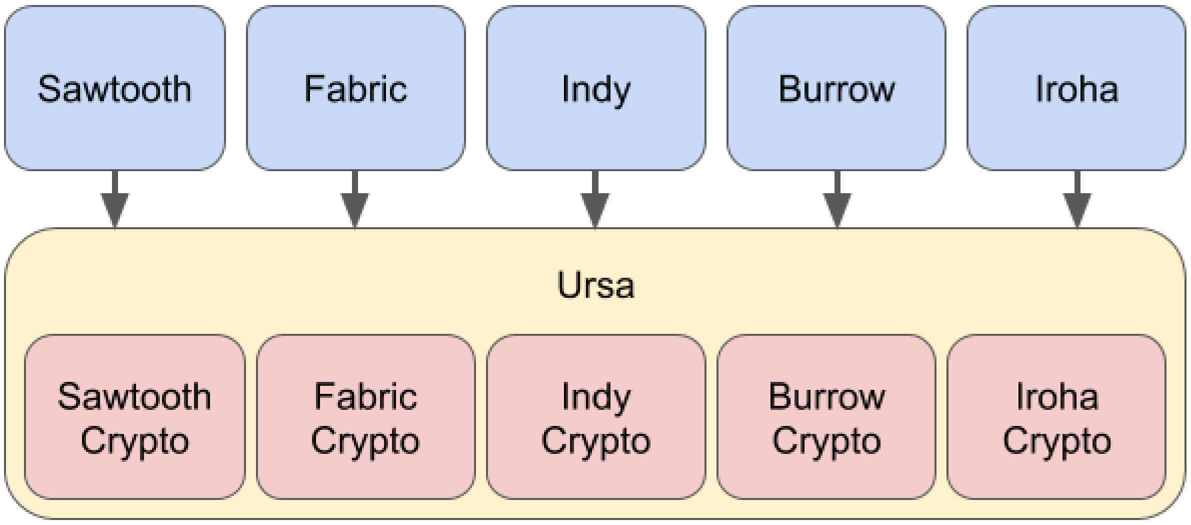

Figure 3 — Ursa prevents duplication.

Point #2: Portability Through Abstraction

All crypto implementations support roughly the same kinds of functions. For instance, most crypto implementations support digitally signing data, encrypting data, decrypting data, hashing data, and generating random numbers. They may use different algorithms with different characteristics, but they all do the same things. I like to say that they all use the same nouns (e.g. crypto keys, signatures, etc) and the same verbs (e.g. hash, encrypt, decrypt, sign, etc). Unfortunately, our blockchain platforms currently interact with their crypto implementations directly which locks them into a single implementation. In software engineering circles, this is called “tightly bound” integration. Tight binding means that if you want to change crypto implementations you have to rewrite the code in the blockchain platform to use the new implementation and not the old. This is because each implementation typically has unique names for their nouns and verbs. For instance, one library may have a hashing function called foo_hash and another will have a hashing function called bar_hash. To change from the foo implementation to the bar implementation, programmers have to do a search and replace from foo_hash to bar_hash. This problem gets multiplied when projects want to use multiple features from multiple crypto implementations.

The solution to the tight binding problem is to write some software that sits between the blockchain platforms and the underlying crypto implementations so that the blockchains can be tightly bound to the in-between code instead of all of the crypto implementations. The in-between code is called an “abstraction layer” by software engineers and allows for multiple different implementations to be hidden from the users of the library, namely the blockchain platforms. Adding an abstraction layer is what is called “loosely bound”.

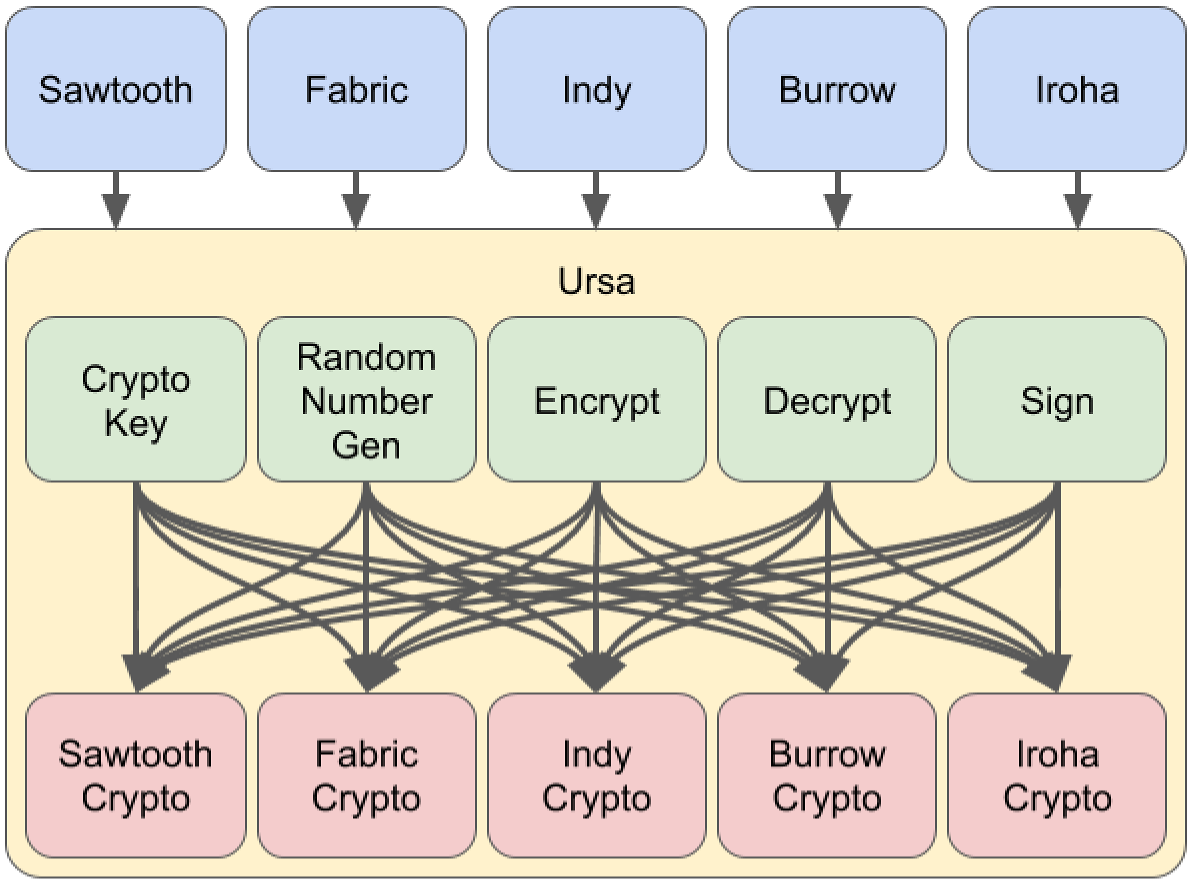

Figure 4 — Abstracting the underlying crypto to present a single interface.

Point #3: Simplification

When writing software that talks directly to a crypto implementation, software engineers have to be very careful to use the library correctly. Different implementations make different guarantees and have different “correct” ways of using them. The foo implementation may have a different way of using it correctly than the bar implementation. This means that engineers have to always remember which library they are talking to and what all of the subtle details and rules are for that library to be used correctly and safely.

One of the largest benefits of writing an abstraction layer to sit between the blockchains and the crypto implementations is that the abstraction layer can be written to use correctly each of the underlying crypto implementations and in turn provide just a single way of correctly driving all of them. The blockchain engineers that use the abstraction layer only have to remember one set of rules instead of a set of rules for each crypto implementation. This decreases the opportunity for making mistakes and also makes it easier for the engineers to audit and review the code that the blockchain platform engineers write.

Point #4: Safety

With Ursa, we chose to write the abstraction layer using a programming language called Rust. Rust is specifically designed for writing secure code. It was created by Mozilla over the last decade to answer the question: “How do we write a truly secure, next generation web browser?” The language and compiler supports features that eliminate all of the most common errors that programmers make. It is also a drop-in replacement for C/C++ and can be used by software written in all other programming languages. This makes it an ideal language for Ursa because we want maximum safety while being usable by Fabric and Burrow, both written in Go; Sawtooth, written in Python; Iroha, written in C++; and Indy, written in Rust.

By choosing Rust for Ursa, our abstraction layer can make certain guarantees that no other programming language can make. By wrapping crypto implementations with Rust, Ursa adds extra safeguards even to implementations written in unsafe languages like C/C++. Ursa can guarantee that memory and data will always be handled correctly and that program execution will always be predictable. All three are crucial for ensuring that the blockchain platforms that use Ursa will have the absolute minimum opportunity for using it incorrectly.

Point #5: Configurability

Another key reason why we need Ursa is to make our blockchain platforms configurable for the situation in which they will be deployed. This is a key feature for being responsive to the demands of our users and to make our blockchains usable in the greatest breadth of scenarios. Cryptography is a unique area of software engineering in that only crypto code has to follow laws regulating implementation and use. For instance, to sell software to the U.S. government, the cryptography has to be FIPS certified. To sell software in China, the cryptography has to only use the algorithms approved by the Chinese government for hashing and encrypting. Most importantly, those two restrictions are mutually exclusive meaning that any software used in China can only support Chinese algorithms and no others. Same is true for software sold to the U.S. government.

The main advantage of Ursa is that blockchain engineers can configure it to only include crypto implementations that meet the requirements of their application. Chinese developers can easily compile Ursa to only have implementations of the Chinese crypto algorithms. The red ex-outs at the bottom of Figure 5 illustrates how compile-time configuration can determine which crypto implementations are included in a build of Ursa. In the case shown in the figure, Ursa was compiled to only include the implementations from Fabric and Indy.

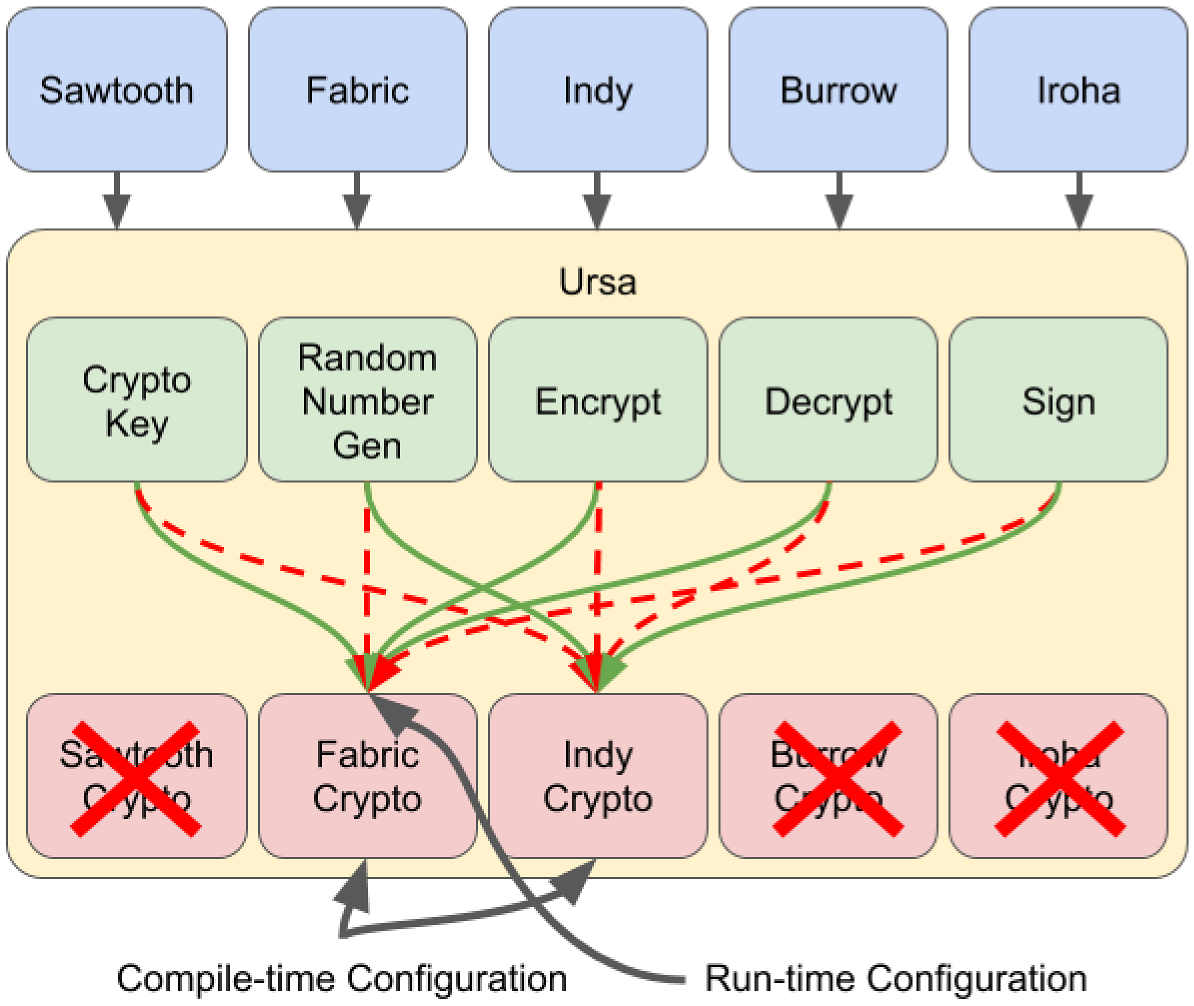

Figure 5 — Ursa supports both compile-time and run-time configuration.

Another main reason we chose to implement the Ursa abstraction layer in Rust is because the project management tool that comes with Rust makes this kind of compile-time configuration very easy to set up and maintain. The tool is called Cargo and it is one of, if not the most, powerful software build tool that exists today. With Cargo, we can provide an easy-to-use and therefore hard-to-mess-up system for specifying exactly which crypto implementations should be included when compiling Ursa.

There is another form of configurability that further motivates this project that is probably the most powerful advantage we're building for our platforms. Ursa supports run-time configuration of the underlying crypto as well. That means the blockchain platforms can change crypto algorithms on the fly as needed. This is key to being ready for when quantum computers get real enough to begin breaking the crypto we are currently using. Once the Ursa maintainers decide that the quantum computer threat is real enough, our blockchain instances can switch from the breakable crypto to the unbreakable crypto without stopping. We've been calling this "algorithmic agility" and it is the Hyperledger answer to the quantum computer question. Figure 5 shows an Ursa built containing crypto implementations from Fabric and Indy and how the run-time configuration is set to use the Fabric implementation for crypto key, encrypting, and decrypting and the Indy implementation for random number generation and signing.

Point #6: Experimentation

The last key reason for Ursa is to give our blockchain engineers the ability to easily experiment with new crypto implementations and protocols. The configurability discussed in Point #5 above means that we can include experimental crypto implementations in Ursa and make them available through the abstraction layer. This makes it possible for blockchain engineers writing applications on Fabric to use the complex signing/proof protocols from the Indy project. It also makes it possible for Iroha to use the secure enclave implementations from Sawtooth.

This sharing of implementations creates an exponential growth in opportunity for blockchain developers using Hyperledger blockchains. It will also make the market for blockchain ideas much more efficient. Our community will be able to quickly find the best combinations of features and architecture for each class of problem. For instance the best solution for tracking electronic health records with a blockchain may be a combination Iroha on mobile devices using the secure enclave encryption from Sawtooth and the crypto proofs from Indy. When Ursa reaches its full potential, that experimental combination, an others like it, will be simple to try out.