Cryptocurrencies need a safeguard to prevent another DAO disaster

High assurance brings mission-critical security to digital funds

MAY 12, 2017 3 MIN READ

TL;DR You want to avoid the next DAO-like disaster: so you want *confidence* that the system underpinning your cryptocurrency doesn't have a hidden flaw that could be triggered at any time and render your assets worthless. To get that confidence you need a *high assurance* implementation of the system operating your cryptocurrency. Formal methods (mathematical specifications and proofs) are the best way to build high assurance software systems, and that is what we are aiming to do with the software behind the cryptocurrencies we build.

How can you sleep at night?

A gold bar or a wodge of cash stashed in a safe has the rather nice property that it doesn't just evaporate overnight. Money managed by computer software is not inherently so durable. Software flaws can be revealed without warning and can destroy the trust in whole systems.

We only have to look around us to see the prevalence of software flaws. The IT trade press is full of news of data breaches, critical security patches, zero-day exploits etc. At root these are almost all down to software flaws. Standard software development practices inevitably lead to this state of affairs.

With the DAO in particular, the flaw was in the implementation of the smart contract that defined the fund, not directly in Ethereum itself. So the implementation of contracts and the design of smart contract languages is certainly an important issue, but the next flaw could be somewhere else. It's hard to know.

So how are we to sleep soundly at night? How can we be confident that our cryptocurrency coins are not just going to evaporate overnight? What we need is assurance. Not to be confused with insurance. Assurance is evidence and rational arguments that a system correctly does what it is supposed to do.

Systems with high assurance are used in cases where safety or a lot of money is at stake. For example we rightly demand high assurance that aircraft flight control systems work correctly so we can all trust in safely getting from A to B.

If as a community we truly believe that cryptocurrencies are not a toy and can and should be used when there are billions at stake then it behoves us to aim for high assurance implementations. If we do not have that aim, are we really serious or credible? And then in the long run we must actually *achieve* high assurance implementations.

In this post we'll focus on the software aspects of systems and how formal methods help with designing high assurance software. Formal methods can be very useful in aspects of high assurance system design other than software, but that'll have to wait for some other blog post.

What does assurance look like?

While we might imagine that assurance is either "yes" or "no" – you have it or you don't – it actually makes sense to talk about degrees of assurance. See for example the summaries of the assurance levels, EAL1 to EAL7, in the CC security evaluation standard. The degree of assurance is about risk: how much risk of system failure are you prepared to tolerate? Higher assurance means a lower risk of failures. Of course all else being equal you would want higher assurance, but there is inevitably a trade-off. Achieving higher levels of assurance requires different approaches to system development, more specialised skills and extra up-front work. So the trade-off is that higher assurance is perceived to come with greater cost, longer development time and fewer features in a system. This is why almost all normal commercial software development is not high assurance.

There are two basic approaches to higher assurance software: the traditional approach focused on process and the modern approach focused on evidence, especially formal mathematical evidence.

Historically, going back to the 1980s and before, the best we could do was essentially to think hard and to be very careful. So the assurance standards were all about rigorously documenting everything, especially the *process* by which the software was designed, built and tested. The evidence at the end is in the form of a big stack of documents that essentially say "we've been very methodical and careful".

Another approach comes from academic computer science – starting in the 80's and becoming more practical and mature ever since. It starts from the premise that computer programs are – in principle – mathematical objects and can be reasoned about mathematically. When we say "reason about" we mean mathematical proofs of properties like "this program satisfies this specification", or "this program always computes the same result as that program". The approach is that as part of the development process we produce mathematical evidence of the correctness of the software. The evidence is (typically) in the form of a mathematical specification along with proofs about some useful properties of the specification (eg security properties); and proofs that the final code (or critical parts thereof) satisfy the specification. If this sounds like magic then bear with me for a moment. We will look at a concrete example in the next section.

One advantage of this approach compared to the traditional approach is that it produces evidence about the final software artefacts that stands by itself and can be checked by anyone. Indeed someone assessing the evidence does not need to know or care about the development process (which also makes it more compatible with open-source development). The evidence does not have to rely on document sign-offs saying essentially "we did careful code review and all our tests pass". That kind of evidence is great, but it is indirect evidence and it is not precise or rigorous.

In principle this kind of mathematical approach can give us an extremely high level of assurance. One can use a piece of software called a proof assistant (such as Coq or Isabelle) which provides a machine-readable logical language for writing specifications and proofs – and it can automatically check that the proofs are correct. This is not the kind of proof where a human mathematician checking the proof has to fill in the details in their head, but the logician's kind of proof that is ultra pernickety with no room left for human error.

While this is perhaps the pinnacle of high assurance it is important to note that cryptocurrencies are not going to get there any time soon. It's mostly down to time and cost, but also due to some annoying gaps between the languages of formal proof tools and the programming languages we use to implement systems.

But realistically, we can expect to get much better evidence and assurance than we have today. Another benefit of taking an approach based on mathematical specification is that we very often end up with better designs: simpler, easier to test, easier to reason about later.

Programming from specifications

In practice we do not first write a specification then write a program to implement the spec and then try to prove that the program satisfies the specification. There is typically too big a gap between the specification and implementation to make that tractable. But it also turns out that having a formal specification is a really useful aid *during* the process of designing and implementing the program.

The idea is that we start with a specification and iteratively *refine* it until it is more or less equivalent to an implementation that we would be happy with. Each refinement step produces another specification that is – in a particular formal sense – equivalent to the previous specification, but more detailed. This approach gives us an implementation that is correct by construction, since we transform the specification into an implementation, and provided that each refinement step is correct then we have a very straightforward argument that the implementation is correct. These refinement steps are not just mechanical. They often involve creativity. It is where we get to make design decisions.

To get a sense of what all this means, let's look at the example of Ouroboros. Ouroboros is a blockchain consensus protocol. Its key innovation is that it does not rely on *Proof of Work*, instead relying on *Proof of Stake*. It has been developed by a team of academic cryptography researchers. They have an academic paper, Ouroboros describing the protocol and mathematical proofs of security properties similar to that which Bitcoin achieves. It has been developed by a team of academic cryptography researchers, led by IOHK Chief Scientist Aggelos Kiayias. This is a very high level mathematical description aimed for peer review by other academic cryptographers.

This is a great starting point. It is a relatively precise mathematical description of the protocol and we can rely on the proofs of the security properties. So in principle, if we could prove an implementation is equivalent (in the appropriate way) to the description in the paper, then the security proofs would apply to our implementation, which is a great place to be.

So how do we go from this specification to an implementation following the "correct by construction" approach? First we have to make the protocol specification from the paper more precise. It may seem surprising that we have to make a specification more precise than the one the cryptographers wrote, but this because it was written for other human cryptographers and not for machines. For the refinement process we need to be more like the pernickety logicians. So we have to take the protocol specification written in terms of English and mathematical symbols and redefine it in some suitable logical formalism that doesn't leave any room for ambiguity.

We then have to embark on the process of refinement. The initial specification is the most abstract and least detailed. It says what must be done but has very little detail about how. If I have a more detailed specification that is a refinement of the initial specification then what that means intuitively is: if you are happy with the initial specification then you would be happy with the new specification. You can have different refinements of the same specification: they differ in details that are not covered in the original specification. Refinement also has a quite specific formal meaning, though it depends on exactly what formalism you're using. In process calculi, refinement is described in terms of possible observed behaviours. One specification is a refinement of the other if the set of possible observed behaviours are equivalent to the other. Formally the kind of equivalence we need is what is known as a bisimulation.

In the case of Ouroboros we start with a very abstract specification. In particular it says very little about how the network protocol works: it describes things in terms of a reliable network broadcast operation. Of course real networks work in terms of unreliable unicast operations. There are many ways to implement broadcast. The initial specification doesn't say. And it rightly doesn't care. Any suitable choice will do. This is an example where we get to make a design choice.

The original specification also describes the protocol in terms of broadcasting entire blockchains. That is the whole chain back to the genesis block. This is not intended to be realistic. It is described this way because it makes the proofs in the paper easier. Obviously a real implementation needs to work in terms of sending blocks. So this is another case for refinement. We have to come up with a scheme where protocol participants broadcast and receive blocks and show how this is equivalent to the version that broadcasts chains. This is an interesting example because we are changing the observed behaviour of protocol participants: in one version we observe them broadcasting chains and in the other broadcasting blocks. The two do not match up in a trivial way but we should still be able to prove a bisimulation.

There are numerous other examples like this: cases where the specification is silent on details or suggests unrealistic things. These all need to be refined to get closer to something we can realistically implement. When do we move from specification to implementation? That line is very fuzzy. It is a continuum, which comes back to the point that both specifications and programs are mathematical objects. With Ouroboros the form of specification is such that at each refinement step we can directly implement the specification – at least as a simulation. In a simulation it's perfectly OK to broadcast whole chains or to omit details of the broadcast algorithm since we can simulate reliable broadcast directly. Being able to run simulations lets us combine the refinement based approach with a test or prototype based approach. We can check we're going in the right direction, or establish some kinds of simulated behaviour and evaluate different design decisions.

There are also appropriate intermediate points in the refinement when it makes sense to think about performance and resource use. We cannot think about resource use with the original high-level Ouroboros specification. Its description in terms of chain broadcast makes a nonsense of any assessment of resources use. On the other hand, by the time we have fully working code is too late in the design process. There is a natural point during the refinement where we have a specification that is not too detailed but concrete enough to talk about resource use. At this point we can make some formal arguments about resource use. This is also an appropriate point to design policies for dealing with overload, fairness and quality of service. This is critical for avoiding denial of service attacks, and is not something that the high-level specification covers.

Of course any normal careful design process will cover all these issues. The point is simply that these things can integrate with a formal refinement approach that builds an argument, step by step, as to why the resulting design and implementation do actually meet the specification.

Finally it's worth looking at how much flexibility this kind of development process gives us with the trade-off between assurance and time and effort. At the low end we could take this approach and not actually formally prove anything, but just try to convince ourselves that we could if we needed to. This would mean that the final assurance argument looks like the following. We have cryptographers check that the protocol description in their paper is equivalent to our description in our logical formalism. This isn't a proof, just mathematicians saying they believe the two descriptions are equivalent. Then we have all the intermediate specifications in the sequence of refinements. Again, there are no formal proofs of refinement here, but the steps are relatively small and anyone could review them along with prose descriptions of why we believe them to be proper refinements. Finally we would have an implementation of the most refined specification, which should match up in a 1:1 way. Again, computer scientists would need to review these side by side to convince themselves that they are indeed equivalent.

So this gives us some intermediate level of assurance but the development time isn't too exorbitant and there is a relatively clear path to higher assurance. To get higher assurance we would reformulate the original protocol description using a proof assistant. Then instead of getting a sign-off from mathematicians about two descriptions being equivalent, we could prove the security properties directly with the new description using the proof assistant. For each refinement step the task is clear: prove using the proof assistant that each one really is a refinement. The final jump between the most detailed refined specification and equivalent executable code is still tricky because we have to step outside the domain of the proof assistant.

With the current state of proof tools and programming language tools we don't have a great solution for producing a fully watertight proof that a program described in a proof assistant and in a similar programming language are really equivalent. There are a number of promising approaches that may become practical in the next few years, but they're not quite there yet. So for the moment this would still require some manual checking. Really high assurance still has some practical constraints: for example we would need a verified compiler and runtime system. This illustrates the point that assurance is only as good as the weakest link and we should focus our efforts on the links where the risks are greatest.

Direction of travel

As a company, IOHK believes that cryptocurrencies are not a toy, and therefore believes that users are entitled to expect proper assurance.

As a development team we have the ambition, skills and resources to make an implementation with higher assurance. We are embarking on the first steps of this formal development process now and over time we will see useful results. Our approach means the first tangible results will offer a degree of assurance and we will be able to improve this over time.