# Good software propagates its own correctness

(with an application to Bitcoin)

By Joannes Vermorel, CEO of Lokad - April 26th 2018

*There are three roads to ruin; women, gambling and technicians. The most pleasant is with women, the quickest is with gambling, but the surest is with technicians.* George Pompidou

Writing computer code is notoriously difficult. Along with software, mankind did invent entirely novel classes of real-life problems ranging from explosions of rockets[1] to hospitals getting ransomed[2]. Those problems - at their core - are the result of badly designed software. This problem has been known for decades, and the software community has been relentlessly improving upon itself on this very subject. The increase of observable software catastrophes is not the result of any kind of *decay* of the software community, but the result of a civilization increasingly dependent - for good reasons - on software. Software problems grow with software usage. Back in 2003, a decade before the Yahoo data breach of 2013[3], a data breach involving 3 billion users would not have been possible because no institution, private or otherwise, had such a user database at the time.

Software *incorrectness* is presently defined as any unintended and undesirable emergent properties of the computer code. The root causes behind bad code ramify in practically every flaw of human affairs. In the following, I will discuss only one of those origins: **not recognizing software code as education material**. This factor is important and yet frequently overlooked by software engineers, busy dealing with the technicalities of their own piece of software. To the credit of those software engineers, technicalities are frequently quite difficult. Thus, even when software engineers are aware of this angle, they don't necessarily have the resources to deal with it.

To see why *education* is involved, let's point out that software code has two distinct audiences. The first audience is the compiler, and later the runtime. The second audience is the peers, i.e. other software engineers.

- Writing code for the compiler is - relatively - easier. The compiler is brutally honest, and it never gets tired of validating and rejecting bad code - the sort of code that does not even compile. The same goes for the runtime.
- Writing code for your peers is vastly more complicated. You need to grasp the point of view of your peers. You need to see the world not only as you understand it, but as your peers misunderstand it as well.

[1] http://www-users.math.umn.edu/~arnold/disasters/ariane.html
[2] https://www.zdnet.com/article/us-hospital-pays-55000-to-ransomware-operators/
[3] https://en.wikipedia.org/wiki/Yahoo!_data_breaches

My main proposition is:

*By educating peers - fellow software engineers - properly written software propagates its inner correctness to the software ecosystem at large by helping peers to replicate, or better, to improve upon, the original correctness.*

As an immediate corollary, this proposition explains why *open source software* is important even for proprietary software companies.

- The propagation of correctness goes both ways. By releasing your code as open source, you get more opportunity to be educated and to improve it further.
- By propagating your *viewpoint* on a given problem through a software product, the software ecosystem *adapts* to make the most of it; that's what *integrations* are about.
- The correctness of your own software is a market signaling mechanism that helps the company to hire further talented software engineers, also seeking education.

Naturally, *open source* is not exclusive with *closed source*; both can coexist within the same company. Time is of the essence too: downsides associated with the release of code as open source decrease over time (unfortunately, upsides decrease as well).

Examples of pieces of software that were not only profoundly correct but who did manage to propagate their own correctness at large are numerous. Let's list three great examples among probably hundreds of equally great examples.

**Lisp**: by popularizing the *functional programming perspective*, this programming language had a distinctive influence on many (most) wildly popular programming languages such as Python, C#, Javascript, etc. Most of the languages that are still being invented nowadays are looking back at Lisp as worthy source for inspiration.

**The C++ Standard Template Library (STL):** a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. The STL succeeded beyond measure in propagating those fundamental data structures. As a testimony to its success, there is no software framework nowadays that does not replicate in one way or another the design of the STL.

**HTTP+HTML combo**: on the surface, HTML looks like a complete Byzantine mess, yet, it is probably the first large scale success of the principle *Be Strict With Yourself, Tolerant Toward Others* applied to software. HTTP+HTML has shown that it was possible to achieve a "good enough" global consensus on seemingly inhumanly complex[4] matters. To a large extend, the

[4] If you think that HTTP+HTML is not inhumanely complex, I suggest to have a serious look at the layout mechanisms involved when badly formatted HTML is found present in a web page while the HTTP download is still *in progress*. A similar feat had already been achieved with the internet itself, but the fine print of HTTP+HTML is orders of magnitude more complex than IPv6.

REST philosophy used for APIs nowadays is largely inspired by the success of HTTP+HTML, merely transferred to the realm of app-to-app interactions.

## Application to Bitcoin

Unlike any of the examples presented above, Bitcoin did seemingly manage the impossible feat of delivering at scale the exact opposite: thousands of *broken-by-design* software initiatives, taking the form of a myriad of digital coins, including quite a few forks of Bitcoin itself.

Instead of educating the ecosystem at large to build upon its own *correctness*, Bitcoin seems to have generated *so far* an endless stream of severe misunderstandings. In my own estimation, Bitcoin is incredibly *correct* in its design, but its correctness certainly did not propagate far, not even to many of its contributors/enthusiasts[5].

Let's briefly review some of the most widespread misunderstandings about Bitcoin that failed to propagate to the ecosystem within the 10 years since the publication of the original Bitcoin paper. None of those problems are terminal though, and as we will see in the following, solutions can emerge, given that proper investments are made.

**No usage quota in Bitcoin, the market force should decide**. A fairly prominent fork of Bitcoin, named Bitcoin Core, did establish a quota on the usage of Bitcoin Core, by capping the block size very low, which resulted in transaction fees spiraling out of control. The block size of Bitcoin is eminently intended to be capped but *only by market forces*. The market will fund the emergence of those solutions as needed. This seemingly simple insight did apparently not propagate to the minds of many of the early Bitcoin contributors.

**Code-is-law is forever impractical, except for Bitcoin itself**. As code-is-law requires to have a near perfect solution for the problem from day 1, it is an incredibly challenging approach to software development. As a result, any code-is-law piece of software itself requires a near-inhuman amount of insights to be done right. Yet, this did not stop hordes of software engineers getting involved with Ethereum and its variants, generating an endless stream of catastrophes - contract breaches - in their trail.

**Small world consensus is required for Bitcoin**. While any variant of Bitcoin does require small world consensus, it certainly did not prevent competing coins from opting for a mesh network instead and failing badly in the process. The most notable attempt is probably the technological trainwreck of IOTA - still part of the Top 10 digital coins as of the time of this writing. IOTA has seemingly exhausted even its critics' the patience, who could not be bothered to list further problems after figuring out the first dozen critical concerns.

[5] I personally made considerable mistakes in my own understanding of Bitcoin - the most notable so far being the *fractional satoshis* angle. I will debunk this incorrect piece of work when I get the time.

**Decentralization means not devolving back to centralization**. Many Bitcoin supporters appear to be missing the point that the Lightning Network (LN) may work, somehow. The main concern with LN is *how it will work*. LN come with economic incentives fundamentally aligned with the present banking system. Thus, it does not really matter whether LN works at all or not, as LN is predictably going to devolve into something essentially similar to the present banking system, merely a lot less secure due to the always-on requirements on the user side.

**Zero-knowledge with Bitcoin means no scalability**. The scalability of Bitcoin fundamentally relies on the transparency of the UTXO dataset. Observers need to be able to tell the difference between an *unspent* entry and an *unspendable* entry. To a lesser degree, observers also need to know *how many satoshis* are left within each UTXO. Unless the digital coin rather radically diverges from the original Bitcoin design, it will not be able to scale and achieve the higher money velocity that it requires to be an economical success *at being money*.

Lacking a better explanation, the impenetrable codebase of the Satoshi client is probably responsible for this situation. The codebase succeeded at delivering a working form of superior money, but due to its opacity, it did not educate the software ecosystem.

To the credit of Satoshi Nakamoto, it was not obvious how his ideas were going to be thoroughly misunderstood by his peers. Thus, Satoshi Nakamoto probably did not invest nearly as much as he should have to preemptively strike down this very problem. Then again, the initial Bitcoin contribution was considerable already. There is a limit to how far you can get a software project while working in stealth mode.

*Fortunately, this insight can also be used to fix the problem, assuming that Bitcoin does not terminally destroy itself while the fix is still under way.*

To illustrate how a codebase can be used to propagate *correctness of understanding*, i.e. to educate, let's take a very narrow engineering problem within Bitcoin. A *narrow* angle is taken for for the sake of concision, however, this approach applies at any scale. Wider angles are simply more difficult and more complex to get right.

Within Terab, the initiative carried by my company Lokad to scale the UTXO dataset of Bitcoin, we are facing the problem of *identifying* Bitcoin blocks. This problem is deceptively simple, and the crux of the challenge *does not* lie in getting it right as far as the compiler or the runtime are concerned - although this is clearly a basic requirement.

The explanation below starts with the *how* and proceeds with the *why*. Unless the reader is familiar with this sort of engineering shenanigans, the whole thing should probably look quite obscure. Please bear with me until I proceed with the explanation of the *why*.

Within Terab, we have opted to identify Bitcoin blocks in three distinct ways:

- *block identifiers*: 32 byte integers externally maintained (block hash)

- *block aliases*: 4 byte integers internally maintained within Terab (increments)
- *block handles*: 4 byte integers externally exposed by Terab (opaque)

A Bitcoin implementation that wishes to use Terab will pass *block identifiers* to Terab, which are then turned into *block aliases* within Terab, and exposed as *block handles* by the Terab API.

A block handle is only valid *as long as the connection to Terab is maintained*. Severing the connection invalidates the handle. Under the hood, within Terab, when a client-server connection is established, Terab generates a pseudo-random bitwise permutation. The permutation is specific to this connection. Before passing any *block alias* to the client, Terab permutates the bits of the *block alias*, and the resulting soup is the *block handle*. An inverse permutation is applied when a *block handle* is passed back to the Terab API to obtain the *block alias*.

The motivation behind the *block identifiers* is simple: hash functions are designed to have extremely low collision rates. Thus, with 256 bits of hashes, Bitcoin should never observe any collision between two blocks sharing the same block identifiers. Conveniently, this also happens to be the canonical way to identify blocks in the Bitcoin ecosystem at large.

The motivation behind the *block alias* is a bit more subtle. As Bitcoin only emits one block every 10 min on average, a 32 bit integer is sufficient to identify thousands of years worth of blocks, including high orphan rates, including discarded temporary blocks built by the miner itself. Within Terab, block aliases are allocated through +1 increments each time a new block is appended to the blockchain. This pattern is convenient to implement various performance optimizations.

The motivation behind *block handles* is arguably downright subtle. Terab should not expose any of its own internal optimization tricks because, by doing so, we would be encouraging software engineers - who should not be assumed to be familiar with Bitcoin - to make predictable classes of mistakes. The bit shuffle is intended (1) to quickly break bad software on top of Terab (2) to avoid misdirecting the community in their understanding of Bitcoin.

Let’s detail the problems that exposing *block aliases* would generate. In Bitcoin, the present orphan rate is below 1%. While the orphan rate may or may not increase with bigger blocks, in all likelihood the orphan rate will stay low. As a result, any engineer who will start working with Terab will observe, through the server logs of Terab or otherwise, the successive block aliases. This engineer would invariably observe that Bitcoin blocks get aliases in +1 increments; orphans are the exception, not the norm. Thus, frequently, the engineer will draw the conclusion: *all blocks can be safely enumerated in +1 increments*. The conclusion is incorrect, but this misunderstanding is going to be **compounded** by the fact that *after 10 blocks or so* the *blockheight* becomes a reliable *incremental* block identifier.

Then, even if the engineer understands that the block aliases are not *exactly* incremental, he/she will observe seemingly “broken” sequences of block aliases such as 100001, 100002, 100004, 100005 which hurt his/her sense of aesthetics[6]. Thus, the engineer will react by trying to “fix” this perceived problem, no matter if it’s a real problem or not. By exposing opaque *block handles*, Terab deprives, on purpose, engineers from observing “elegant” sequences of identifiers and drawing incorrect conclusions from them.

A project like Terab - if it ever enjoys any degree of success - can accidentally shape *incorrect opinions* about Bitcoin. While it can be argued that reducing the variance of the miners payout is a good thing ... or not, Terab should certainly do everything it can to avoid accidentally polluting this debate with its own technicalities.

Conversely, by exposing 4 byte identifiers for blocks, Terab can also be used to shape *proper opinions* about Bitcoin. For example, Terab could emit warnings if less than 1% of the block identifiers pushed to Terab make their way, permanently, into the longest blockchain. If the vast majority of the blocks that are pushed to Terab do not make it into the longest blockchain - to the point that the allocation rate would endanger the 32 bits capacity of the block handles - then most likely one or several engineers are abusing Terab and misunderstanding Bitcoin in the same process. Terab should prevent this situation from happening by failing or quasi-failing *on purpose* as a way to help peers in gaining the necessary knowledge about Bitcoin. Failing fast is better than failing late.

In conclusion, software that happens to be both good technical and educational material can propagate correct beliefs into the software ecosystem and foster innovation in its trail. If the software happens to be technically good, but to fail on the educational angle, then its success is likely to be slowed down by recurrent misunderstandings. Furthermore, the more the software succeeds technically without educating accordingly, the more disorder is induced in the software ecosystem.

Bitcoin is a good example of this proposition. While being an incredible success, Bitcoin appears to have failed - so far - at educating the software ecosystem about its own internal correctness. Yet, as illustrated by a simple example, like most of the other problems faced by Bitcoin in the past, it’s a fairly solvable problem given that the proper efforts get injected into its resolution.

---

[6] Aesthetics in software engineering are very important. It’s one of the most powerful heuristic to detect incorrect code. Experienced software engineers only need a cursory glance at a codebase to get a sense of whether this codebase is a mountain of problems waiting to happen (the code is ugly), or a seemingly dependable product (the code flows organically).