Reflections on Trusting 'Trustlessness' in the era of "Crypto"/Blockchains

Evangelos Georgiadis*

November 8, 2021

> #Crypto represents a generational shift to trust in math and logic over trust in human behavior. #BTC#EOS #ETH
> Brendan Blumer (2021/01/13)

Trust is a funny word, noun and concept, (see `https://www.merriam-webster.com/dictionary/trust`), particularly in the realm of code or programs. Trust, or for that matter, trustlessness, is a key value proposition in a field that deserves Peter A. Thiel's *0 to 1 innovation status* (see `https://www.amazon.com/Zero-One-Notes-Startups-Future/dp/0804139296`), namely, decentralized Finance, (DeFi), or as block.one attempts to distinguish, "compliant open source Programmable Finance" (ProFi), where computation supposedly happens in a "completely trustless and secure manner".

This key proposition deserves a more granular, if not critical, inspection, not just because notionally, (in cumulative terms) trillions of USD are residing on blocks and tokens (see `https://coinmarketcap.com/charts/`), but because theoretical computer science might offer

1. insightful perspectives or a constructive framework to think and reason,
2. useful toolkits, and
3. a richer understanding of intrinsic computational limitations/properties that might impact "trustlessness"

'Trust' and 'security' are properties that have often been discussed and analyzed on a protocol level and, in particular, the consensus level. The reader might be interested in Vitalik Buterin's (more generic, yet still useful) take on *Trust Models* (see `https://vitalik.ca/general/2020/08/20/trust.html`).

We argue that at the most granular level, the user remains at the mercy of the "human element" which, in some capacity and to some degree, has been hardcoded into the codebase and thus protocol. (NB: A poor yet instructive analogy might be the theme of bias in ML/AI algorithms.) Additionally, even though the codebase might be open source, and thus inspectable, validating integrity of a million-loc-long codebase isn't something that the average

*CQuant Technologies Limited: `egeorg@cquant.xyz`

user is capable of. At last, even if theoretical computer science has toolkits to offer for automatically detecting obvious deficiencies either via formal verification methods, or employing our cool friends at *Runtime Verification Inc* (see, `https://runtimeverification.com`), the problem of fixing bugs at the protocol level, in a decentralized setting, morphs into a non-trivial coordination problem where a lot of human behavior is necessitated – see Neha Narula et al's article *Responsible Vulnerability Disclosure in Cryptocurrencies* (see `https://dl.acm.org/doi/10.1145/3372115`).

Fixing bugs at that level might require hardforks, which in turn begs the very essence of trustlessness.

In the spirit of Ken L. Thompson's (1984) Turing Award Lecture, *Reflections on Trusting Trust* (see `https://dl.acm.org/doi/10.1145/358198.358210`), we would like to reflect on who and what we should trust, is it ... (NB: We employ a three levels of analysis approach to put things into perspective: actors, software/hardware level, systemic level.)

1. (actors) the architects that came up with the foundational blue-prints (aka conceptual proof: protocol/consensus mechanisms)?
2. (actors) the software engineers that implemented the code (for the various layers)? or the
3. (software/hardware) the actual language that is supposed to help us – the users – (efficiently and accurately) communicate with the machines or nodes?
4. (systemic) intrinsic limitations/properties that arise from sufficiently sophisticated computation?

(1) – (2): Actors can be classified as being either myopic or corrupt but never omniscient! (there's a fine line between these two – almost semantics). Myopic architects or software engineers leave behind "bugs", whereas corrupt software engineers or architects engineer "features".

For example, a corrupt architect (be it a nation state, or someone incentivised to go rogue) will aim for conceptual corruption – providing a perfectly sound solution with exploitable properties. (This is the type of exploits that goes potentially unnoticed via any formal verification method of a codebase. Additionally, represents a more sophisticated variant of Ken Thompson's "Trojan horse" concept.) A vivid primer of conceptual corruption is Markus G Kuhn's 1 page, 3 column article entitled *Backdoor Engineering* (see

`https://cacm.acm.org/magazines/2018/11/232225-technical-perspective-backdoor-engineering/fulltext`).

Kuhn talks about saboteurs infiltrating teams and luring them to constructing sound machinery – 'provable security based on number-theoretical assumptions' – with exploitable properties. Of course, our very good and diligent friends at the *NSA* (see `https://www.nsa.gov`) are working towards that direction focusing on a few cryptographic key primitives – causing some blockchains such as Ethereum to employ KECCAK-256 which does not follow the FIPS-202 based standard (aka SHA-3) that was finalized in 2015.

On the other hand, a myopic architect might not possess sufficient insights to forsee potential risks impacting "trustlessness" and go with what is convenient and readily available.

(3): The question of whether TCS offers any toolkits that can detect conceptual correction is unknown to this author, but probably known to our friends at Fort Maede.

Theoretical computer science has many toolkits to offer in regard to formal software verification, the most prominent being, *Coq* (see `https://coq.inria.fr/`) and *HOL* (see `https://hol-theorem-prover.org/`).

It is also delightful to see how our friends from Algorand (see `https://www.algorand.com`) employed these technologies to verify a certain property of Algorand's consensus mechanism (see `https://runtimeverification.com/blog/formally-verifying-algorand-reinforcing-a-chain-of-steel-modeling-and-safety/`). That said, obvious trust issues with these toolkits remain, and have been highlighted in Donald MacKenzie's book *Mechanizing Proof: Computing, Risk, and Trust* (see `https://mitpress.mit.edu/books/mechanizing-proof`).

(4): Finally, and perhaps more interestingly, on the systemic level, the question of how trustlessness can be exploited in a computationally universal setting (i.e., assuming that a given consensus protocol turns out to be computationally universal in some notion) should be explored.

The moral of the story might not be as obvious as Ken Thompson led us to believe, namely,

> You can't trust code that you did not totally create yourself.(p.763)

A more trustworthy but less gratifying conclusion might be Don Knuth's famous line:

> Beware of bugs in the above code; I have only proved it correct, not tried it.