Smarter Signatures: Experiments in Verifications

A White Paper from Rebooting the Web of Trust II: ID2020 Design Workshop

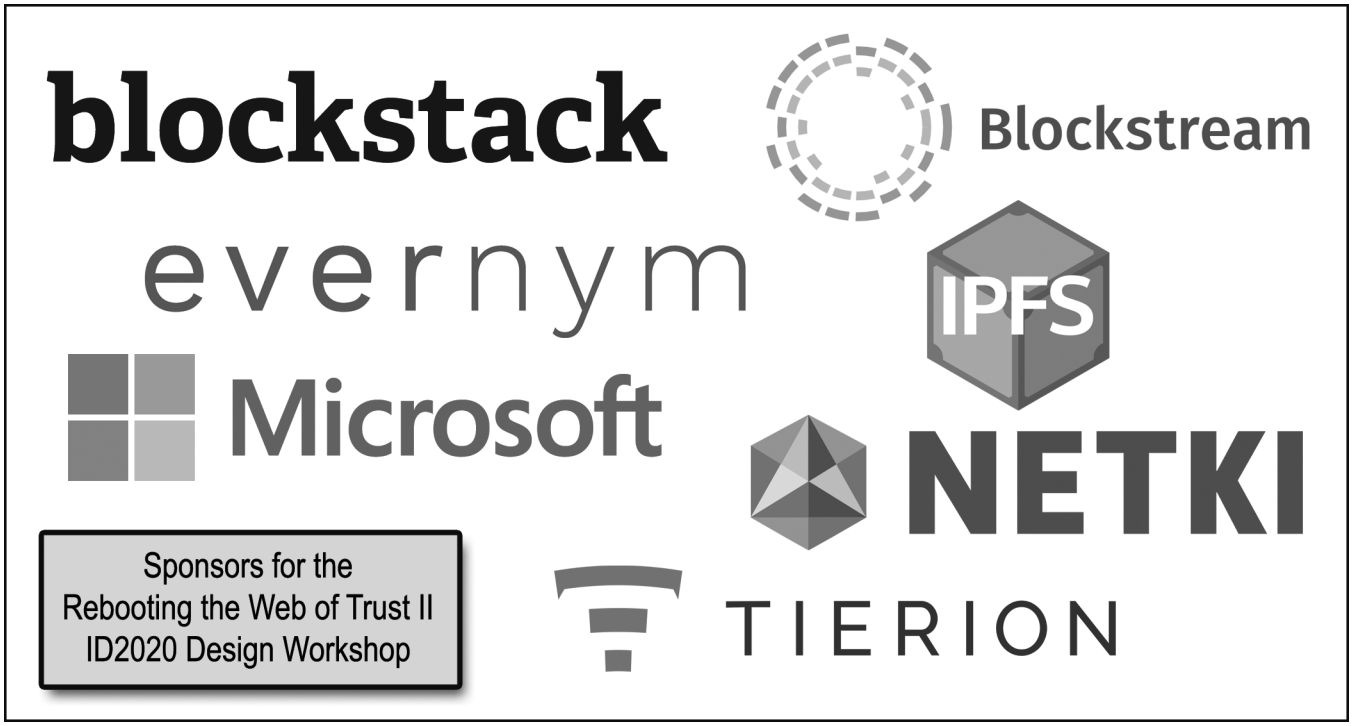
By Christopher Allen & Shannon Appelcline

ABSTRACT

Technologies like the Web of Trust and PKI lay the foundation for *identity* on the internet: they map a human persona to a cryptographic construct that is represented by a public key and protected by a private key. *Digital signatures* are fundamental to these digital identities and have been widely used in a variety of applications. They're the heart of SSH, the foundation of certificates, and the core of newer technologies like blockchain.

However, today's simplistic signatures are just the start; they can be improved, to create more powerful and more complex signatures that can truly be better and *smarter*.

Now is the time to begin experimenting with these possibilities.



1. AN OVERVIEW OF SIGNATURES

The traditional usage of digital signatures is quite straightforward. The owner of a cryptographic identity signs a message (or a certificate) with his private key; a recipient can then use the related public key to verify the message.

Bitcoin is one of the few technologies that offers something more: multisignatures. Transactions can be signed by up to N different people, of whom M are required; for example a 2 of 3 multi-sig would require two people from a group of three to sign a transaction.

However, Bitcoin multi-sigs are just a first step. Even if the simple signatures of the modern day were expanded to include multisignatures of any size, they still wouldn't support the full richness of business and computer logic that is becoming a part of our digital life. Simple signatures can't offer the flexibility that is needed by modern enterprises, and they can't offer the reliability that is required for modern finances.

To support these needs requires a new kind of signature — a *smarter signature* that increases options while still meeting the responsibilities of a robust and trusted signature system.

2. THE USES OF SMART SIGNATURES

The core use of a signature is *verification*: it must ensure that the authorization conditions required for a task are met. In the world of simple signatures, that meant verifying that the right person signed a message. However smart signatures have a wider scope, supporting many more use cases.

Some examples follow. They should be considered a starting point, enumerating some of the needs for smart signatures, without being a be-all or end-all.

1. ***Multifactor Expressions.*** A smart signature should support the inclusion of multiple elements within a single signature.
    1. ***Multisignature Expressions.*** A smart signature should support the inclusion of multiple signatures via a logical AND operator. The signatures would *all* be required for verification, forming an N of N multisignature, such as joint homeowners who all need to sign over a deed.
    2. ***Multisignature Subsets.*** A smart signature should also support M of N multisignatures where only *some* signatures are required for verification, such as a married couple, either of whom can write checks from a joint bank account.
    3. ***Multisignature Equivalents.*** A smart signature should support the inclusion of multiple signatures via a logical OR operator, such as when a legacy RSA signature, a current EC signatures, and a future-proof Hash Signature (which is quantum resistant but very slow) are all included and any of them can be used to verify the signature.
    4. ***Varied Content.*** A smart signature should support the inclusion and combination of a variety of different signature elements, including other verification elements like biometric signatures and proof of hardware control, such as a lead developer who requires both his signature and a hardware token to sign off on software releases. It should also support the inclusion of elements helpful to enable delegation and other signature uses, such as timestamps.
2. ***Signature Delegation.*** A key holder should be able to precisely control how his key and his signature are used.
    1. ***Time-Limited Delegation.*** A key holder should be able to authorize a person or a device to sign for a limited time, such as when the key holder is on vacation or at a conference.
    2. ***Time-Expired Delegation.*** A key holder should be able to automatically authorize a person or a device if their own use of a key goes inactive for an extended amount of time, such as when a key holder dies, and their successor needs to take over signatures.
    3. ***Use-Limited Delegation.*** A key holder should be able to authorize a person to sign only in limited situations, such as a software team that can sign a development version of software but not a stable version.

4. ***Content-Limited Delegation.*** A key holder should be able to authorize a person to sign messages with specific content, such as a financial department that can sign to issue bonds with a maximum amount and a bounded interest rate.

5. ***Non-Signature Delegation.*** A key holder should be able to authorize a person to use a key in certain instances, such as to revoke a key or to revoke a previously signed certificate, but not to sign documents.

3. ***Internal Depth.*** A smart signature should support internal depth by combining these different possibilities, such as a development release of software that includes both multifactoring and delegation by requiring 3-of-5 signatures, where one signer has authorized his assistant because he's on leave, and another signer requires 2-of-2 keys for his signature, one of which is stored on a hardware token. Because this depth is created through internal links, the requirements are all evaluated synchronously.

4. ***Transactional Support***. A smart signature *system* should support external depth by being able to prove that specific states have been reached in a larger state machine through the chaining of multiple signatures, such as when an art dealer needs to examine the transactional history of a painting to ensure that he's not purchasing stolen goods. Because this depth is created through external links, the requirements tend to be evaluated asynchronously: one smart signature at a time.

These use cases all focus on the *creation* of signatures, providing functionality that signers need. However, there are actually two users for any signature: the signer and the verifier. Additional verifier-focused use cases may illuminate UI and UX requirements for a smart signature system.

Some of these use cases obviously will require some calculation. However, smart signatures are ultimately about verification, not computation. In fact, our suggested requirements for smart signatures do their best to minimize computation as a factor for any verifying user.

3. THE REQUIREMENTS OF SMART SIGNATURES

Because smart signatures offer increased complexity over simple signatures, care must be taken to ensure that the complexity does not overpower the security of either the signatures or the systems that they're running upon. To ensure this, six requirements are listed, as suggestions for smart signature systems:

1. ***Composable.*** The increased complexity of smart signatures requires that they be built using some sort of programming language. However, the language itself must remain simple, with complexity built up from a constrained set of operations. This ensures the security of the signature language.

2. ***Inspectable.*** Signatures must be easily understandable by a qualified programmer, so that any sophisticated user can readily evaluate the elements of a signature and how they will be verified. This requirement often emerges naturally from composability; it ensures the security of the signatures, with a focus on human-driven security.

3. ***Provable.*** Signatures must be formally analyzable, so that they can support logical reasoning and so that sophisticated users and expert computer tools can have foreknowledge of the requirements of verification. This further supports the security of the signatures and also foreshadows support for the security and stability of the computer systems.

4. ***Deterministic.*** Signatures must always produce the same results, even when run on different machines or different operating systems. This also ensures the security of the signatures, but it focuses on machine-driven security.

5. ***Bounded.*** Signatures must not be able to exceed appropriate CPU or memory limitations through creation of malicious (or bad) signings. They need to minimize their size in order to minimize bandwidth and storage costs. Additionally, enforcement of these limitations must be deterministic. This also ensures the security of the computer system.

6. ***Efficient.*** Though we place no requirements on the difficulty of creating signatures, the cost of verifying them should be very low. This also ensures the stability of the computer system.

One other element that should be considered is ***privacy***. In smart signature design, there is a trade-off between flexibility and fungibility: many of the functions that make signatures smarter also require participants to reveal more about who they are, reducing the substitutability of the persons involved in the signatures and of any

resources being signed. Even if privacy is not a *requirement*, it should be a *consideration*; any decisions about the level of privacy in a signature system should be known and purposeful.

A smart signature system that supports the use cases described above, that meets the requirements listed here, and that considers its privacy implications, would add powerful tools to the digital world by meeting the needs of the financial and business worlds.

4. EXPERIMENTING WITH SMART SIGNATURES

Fulfilling these uses and meeting these requirements for smart signatures necessitates the creation of better languages and better tools. However, the creation of a new foundation for smart signatures (and eventually smarter contracts) can be tricky and full of pitfalls, as shown by the recent problems plaguing The DAO on Ethereum, where flaws in a contract's code led to the theft of tens of millions of dollars[1].

The Ethereum crisis clearly shows that the design of new languages for smart signature systems must be thorough and comprehensive. Architects must experiment with many options, to ultimately produce something that is stable and trustworthy.

A few different possibilities are discussed below. They should be considered starting points, not ending points. They are not being offered as standards, nor even as the preferred options for smart signature systems. They are instead offered for discussion and for expansion, in the hope that they will eventually lead the way to a more robust smart signature system and the beginning of a more robust web of trust.

4.1 The Languages of Smart Signatures

Functional programming languages are a good choice for the foundation of smart signatures because they meet three of the suggested requirements: they're composable, they're provable, and they're deterministic. The composability and provability emerge from the fact that functional programming languages are built of pure mathematical functions; it's easy to put them together to create more complex systems and it's easy to prove what they do. The determinism emerges from the fact that functional programming languages do not support state or mutable data; they guarantee that the same inputs will always produce the same outputs.

There are a few options for functional languages. Lambda calculus languages are the classic choice, but the Forth-like Bitcoin Script with its stack-driven functionality offers another possibility. More far flung options are also considered, such as the logical sequent calculus.

4.2 Experiment #1: Bitcoin Script

One option for building a new smart signature system is to start with something that already exists and that is already being used to safeguard millions of dollars worth of transactions: Bitcoin Script[2].

Bitcoin Script currently authorizes the spending of Bitcoins. Typically, each Script is linked to either a single signature or else to a M-of-N multi-sig. However, it's also possible to encode more complex redemption conditions into a Bitcoin Script, and even to keep them secret — allowing a recipient to prove that he met the signing conditions by matching a hash of those conditions.

Though Bitcoin Script is currently used on the blockchain, this is not a requirement. The robust signing language could be used outside of the blockchain — protecting other sorts of authorization systems and creating a generalized smart signature language.

Though Bitcoin Script is constrained, the following example shows that it's nonetheless a robust functional language:

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

This simple script checks a signature against a public key in order to verify the signature. Much more complexity is possible.

Advantages: The biggest advantage of the Bitcoin Script approach is that it's *well-tested.* It's been in use for almost a decade with a focus on authorization, which means that it's already laid the groundwork for a smart signature system.

It's also *well-trusted.* Bitcoin Script is the heart of the Bitcoin system, making it the ultimate guardian of a $10 billion market cap. It's a language that has proven its financial responsibility.

Finally, it's *constrained.* Though Bitcoin Script contains an extensive menu of operations (opcodes), they've been curated: some opcodes were disabled in Bitcoin's early days to prevent mischief. Because of these constraints, the individual elements of a Bitcoin Script can be examined in isolation, making it easy to see any problems. That's

much of what's maintained the trust in Bitcoin Script and ensured that there were no DAO-style crises.

Disadvantages: The advantageous constraints of Bitcoin Script may also create one of its biggest disadvantages: it's *limited*. In fact, Bitcoin Script may be too limited to offer the full menu of options required for smart signatures. However, there are already Bitcoin Improvement Proposals in place to increase the set of Bitcoin Script opcodes[3] [4], while Bitcoin's new segregated witness (segwit) support will make future changes to Bitcoin Script even easier[5]. Blockstream's sidechains[6] offer an alternative: incorporating new operations without changing the original Bitcoin blockchain.

Removing Bitcoin Script entirely from the blockchain offers another way to enable language updates, but it also raises another issue: Bitcoin Script is currently *locked to blockchains*. Though it may be possible to use it independently, this has not been tested and may raise future issues of compatibility.

Finally, Bitcoin Script is a Forth-derived language, which means that it is *stack-oriented*. This requires a particular type of logic that may make it harder for some people to parse or understand — though this may also be the case for fully functional languages like lambda calculuses or more outré languages like those based on sequent calculus.

4.3 Experiment #2: Dex

Peter Todd is working on another possible system for smart signatures, one that he calls Dex, a system of deterministic predicate expressions[7]. Much like Bitcoin Script, Dex's predicate expressions evaluate simply to either true or false results. However, the other part of Dex's name is just as important: it's deterministic, guaranteed to always return the same result for a specific signature and environment.

Dex also more fully embraces functional programming: it's built using lambda calculus. As with Lisp, atoms of numbers, strings, symbols, and cells are recursively built up into s-expressions. (In other words, Dex contains parenthesized lists that regularize and order the evaluation of functions.) These s-expressions are then merkelized (hashed), producing unique digests.

Dex expressions should look quite familiar to Lisp programmers:

```
(sig_valid <pubkey> <sig> <hash>)
```

This function might be accessed with a lambda function like the following:

```
(sig_valid <pubkey> (cdr argm) (sha256 (car argm)))
```

Which allows a message and its signature to be passed into the sig_valid function:

```
(sig_valid <pubkey> (cdr '(<msg> <sig>)) (sha256 (car '(<msg> <sig>))))
```

When the cdr, the car, and the sha256 hash are all evaluated, the sig_valid function can then do its job and determine the validity of the signature. Again, much more complexity is possible.

Advantages. One of the biggest advantages of Dex is one of its core features, its *determinism*. The ability to run code on different computers and get the same results is vital for the consensus of any signature system. With a language like Bitcoin Script, which is not built on an entirely functional language, this sort of determinism was much more difficult to achieve. In Dex, thanks to its basis in lambda calculus, it's there from the start as an integral feature of the language.

Dex also has excellent properties of *efficiency*. The use of a Merkle tree helps Dex to enable pruning: unneeded data in an expression can be cut out and replaced with hash digests, making it easier to use lite clients.

Finally, Dex is *upwardly mobile*. Smart signatures can be building blocks for creating full smart contract systems, and this is an option that Todd has considering from the start. He even calls Dex one of the "Building Blocks of the State Machine Approach to Consensus", with its deterministic expressions being states in the state machine[8].

Disadvantages. Lambda Calculus is generally an *unusual* sort of programming language. It doesn't have the same coding styles or the same programming patterns as more common, imperative languages. This is what enables many of its desired features, but it can also prove a disadvantage to some programmers. Further, languages like Common Lisp and Scheme have long been used in entry-level college computer classes, which may create cognitive biases in students who were confused about the unusual though processes required.

Compared to a well-tested language like Bitcoin Script, Dex is quite *novel*. It's not just untested, it's truly experimental. Though there's great potential for its expansion, it will have to be thoroughly examined before it can reach that potential.

4.4 Experiment #3: Crypto Conditions

Crypto-conditions[9] were developed by Stefan Thomas as part of the Interledger[10] project, based on a requirement for a smart signature data type in the Interledger Protocol's core data model. The protocol relies on one or more ledgers that are involved in an end-to-end transfer being able to put funds on hold pending the fulfillment of a predefined condition. This condition is, in effect, the definition of a smart signature and the fulfillment of that condition is the signature itself. More information is available from Crypto-condition workshops presented at Interledger[11] and at IETF[12]. In adition, Crypto-Conditions (draft-thomas-crypto-conditions-00) has been submitted as an Internet Draft for candidacy as a standards track RFC[13].

An essential requirement of crypto-conditions is that any implementation *must* be able to evaluate if it will be able to validate the signature later (fulfillment) just by looking at the signature definition (condition). This allows a ledger to reject a transfer that is using a condition the ledger doesn't support before the end-to-end transfer is fully prepared, avoiding a case where the ledger fails to release the funds upon receipt of the signature (fulfillment) because they are unable to validate it. It also meets the core purpose of the provability requirement for smart signatures, even if it does so by a slightly different manner.

Crypto-conditions define a format for encoding these signature definitions (conditions) and signatures (fulfillments) that incorporates versioning, a feature-requirement bitmask, and a max-fullfilment size requirement. This supports validation of the fulfillment conditions and offers other advantages ...

Adventages. Crypto-conditions are *deterministic.* Rather than attempting to define a Turing complete signature language crypto-conditions simply combines existing primitives that can be deterministically validated on any platform. As such, the combined result, which uses simple boolean algebra, is also deterministic across platforms.

Crypto-conditions are also nicely *compact.* Complex boolean logic trees of hashed conditions can be compacted down to a single hash using Merkle trees, while a fulfillment can also leave any unfulfilled branches (such as in an m-of-n signature) as hashes.

Disadvantages. Crypto-conditions is another *novel* system that is still undergoing development.

4.5 Experiment #4: Sequent Calculus

Russell O'Connor offers a fourth approach to smart signatures based on sequent calculus. This approach envisions smart signatures with formal proofs, where simpler proofs are functionally combined to ultimately create smarter signatures that are analyzable formally. The type system limits the sequent calculus to defining only finitary functions with bounded complexity, while the language comes with formal semantics that are easy to define in software proof assistants. A full paper on this topic is pending.

Advantages. The best advantage of a sequent calculus is that the formal semantics can be *formally reasoned* about, and programs can be proved correct using software proof assistants. Furthermore, the interpreter for the sequent calculus can also be proved correct, potentially allowing for an end-to-end proof of correctness "down to the metal".

Disadvantages. The main disadvantage of a sequent calculus approach is that it's perhaps even more *esoteric* than the state machines and lambda calculuses previously described. There will likely be some issues with inspectability as a result. However, it's possible that a language could be built atop the formal proofs that made them more accessible.

5. LESSONS LEARNED FROM OTHER PLACES

There are numerous lessons that can be learned for smart signatures from other places — particularly lessons related to security. As the Ethereum crisis showed us, smart signatures and smart contracts won't be secure until their programming languages are secured and protected against errors. Resolving this problem is just as important as laying the foundations of a smart signature language. Fortunately, a number of people have been tackling this issue.

Jack Pettersson and Robert Edström of the Chalmers University of Technology have written a thesis on making smart contracts safer[16]. Their approach focuses on Idris, a functional programming language with lambda binding. It uses an advanced type system to offer solutions for several classes of common errors and even provides a backend for Ethereum. More broadly, the SecLang task force[17] focuses on security in programming languages. They have been writing papers for decades that analyze security, improve privacy, and remove vulnerabilities. Their approaches could be vital to enabling that same security in smart contract languages. Both Chalmers and SecLang point us toward options for safety and security in

smart signatures languages of all sorts; there are doubtless other possibilities.

However, many of these approaches also have their origins in other fields. Bitcoin and Interledger both have strong roots in the internet community and can offer lessons on how existing payment networks will need to interact with new smart contracts. Meanwhile, existing papers on sequent calculus[14][15] can offer a foundation for O'Connor's unique and innovative approach. The past is prologue, but its lessons learned are important for moving into the future.

Open Questions

Figuring out how to create and secure a new language for smart signatures is just the first step. There are many other open questions, some of which were raised in an earlier Rebooting the Web of Trust paper[18].

Context. Though functional languages are stateless, they still require contexts: how do they receive input? Bitcoin provides context through the output script of a previous transaction. Other online tools have internal contexts, external contexts, or run contexts. However, there's nothing parallel for smart signatures. What contexts are required, and how should they be implemented?

Revocation. How do we allow signers to revoke a signature? Do we need to separate out proof of validation and proof of non-revocation in a script? Is it even possible to *prove* non-revocation? Alternatively, should we severely limit the lifespan of signatures to avoid the question of revocation entirely?

Hierarchical Deterministic Keys. Some use cases, such as short-term delegation, could benefit from Hierarchical Deterministic Keys (HDKs), where children key can be created from a parent key. How can these HDKs be incorporated into smart signatures and how can they be secured?

Oracles. A third-party oracle can help with the evaluation of certain conditions such as proof-of-existence and perhaps even proof-of-non-revocation. However, oracles may also be what separates smart signatures from more complex smart contracts. Does a simple subset of oracles have any place in the world of smart signatures? If so, what's the actual dividing line between a smart signature and a smart contract?

6. CONCLUSION

This paper is meant to be an icebreaker. Though it offers some suggested smart signature use cases and requirements, they're evolutionary. They were incorporated in part from the Rebooting the Web of Trust I "Smart Signatures" (2015) paper and in part from Peter Todd's "Dex: Deterministic Predicate Expressions for Smarter Signatures" (2016) paper, then they were expanded and reorganized for this paper. In other words, they're works in progress that could still benefit from additional input. Similarly, the experiments summarily reviewed in this paper are just four of many. More possibilities and more discussions are welcome!

Smart signatures are an important tool that could change the way business is done on the internet; they could revamp how we live and even play in electronic communities. As a result, it's critical that we get them right, that we not repeat the mistakes of The DAO and other sophisticated computer systems that went before us, but which didn't live up to the rigors of actual usage.

So we offer this paper as the beginning of a conversation about how to create *smarter* signatures.

What do you suggest?

FOOTNOTES

[1] Del Castillo, Michael. 2016. "The DAO Attacked: Code Issue Leads to $60 Million Ether Theft". Coin Desk. *http://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/*.

[2] Allen, Christopher, Greg Maxwell, Peter Todd, Ryan Shea, Pieter Wuille, Joseph Bonneau, Joseph Poon, and Tyler Close. 2015. "Smart Signatures". Rebooting the Web of Trust I. *https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust/blob/master/final-documents/smart-signatures.pdf*.

[3] Torpey, Kyle. 2016. "New BIP Would Enable Better Privacy, CrossBlockChain Exchange, TrustFree Betting, and More for Bitcoin". CoinJournal. *http://coinjournal.net/new-bip-enable-better-privacy-crossblockchain-exchange-trustfree-betting-bitcoin/*.

[4] Lau, Johnson. 2016. "BIP 114. Merkelized Abstract Syntax Tree". GitHub. *https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki*.

[5] Bitcoin Core. 2016. "Segregated Witness: The Next Steps". Bitcoin Core. *https://bitcoincore.org/en/2016/06/24/segwit-next-steps/*.

[6] Maxwell, Gregory. 2015. “Extending Bitcoin with Sidechains”. Blockstream. https://blockstream.com/developers/.

[7] Todd, Peter. 2016. “Dex: Deterministic Predicate Expressions for Smarter Signatures”. Rebooting the Web of Trust II: ID2020 Workshop. https://github.com/WebOfTrustInfo/ID2020DesignWorkshop/blob/master/topics-and-advance-readings/DexPredicatesForSmarterSigs.md.

[8] Todd, Peter. 2016. “Building Blocks of the State Machine Approach to Consensus”. Peter Todd. https://petertodd.org/2016/state-machine-consensus-building-blocks.

[9] Thomas, Stefan. "Crypto Conditions". GitHub. https://github.com/interledger/rfcs/tree/master/0002-crypto-conditions.

[10] Interledger web site. https://interledger.org.

[11] Thomas, Stefan. 2016. "Crypto-conditions". ILP Workshop. https://www.youtube.com/watch?v=YfBDDWp58po&list=PLIR1FI1vEGeGvladEm-YZIvokXyH4bbIL&index=7.

[12] Thomas, Stefan. 2016. "Crypto-conditions: A Standard for Composable Signatures". IETF96 Ledger. https://youtu.be/uPXXfClTqSY?t=49m8s

[13] Thomas, Stefan. 2016. "Crypto-Conditions: draft-thomas-crypto-conditions-00". IETF Datatracker. https://datatracker.ietf.org/doc/draft-thomas-crypto-conditions/

[14] Ariola, Zenn M., Aaron Bohannon, and Amr Sabry. 2009. "Sequent Calculi and Abstract Machines". ACM Transactions on Programming Languages and Systems. http://www.cs.indiana.edu/~sabry/papers/sequent.pdf.

[15] Guenot, Nicolas and Daniel Gustafsson. 2015. "Sequent Calculus and Equational Programming". IT University of Copenhagen. http://arxiv.org/pdf/1507.08056.pdf.

[16] Edström, Robert and Jack Pettersson. 2016. “Safer Smart Contracts through Type-Driven Development”. Chalmers University of Technology. http://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf.

[17] SecLang Taskforce. 2016. “Security: Programming Languages”. DistriNet. https://distrinet.cs.kuleuven.be/research/taskforces/showTaskforce.do?taskforceID=seclang.

[18] Allen, Christopher, Greg Maxwell, Peter Todd, Ryan Shea, Pieter Wuille, Joseph Bonneau, Joseph Poon, and Tyler Close. 2015. “Smart Signatures”. Rebooting the Web of Trust I. https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust/blob/master/final-documents/smart-signatures.pdf.

MAJOR REFERENCES

Allen, Christopher, Greg Maxwell, Peter Todd, Ryan Shea, Pieter Wuille, Joseph Bonneau, Joseph Poon, and Tyler Close. 2015. “Smart Signatures”. Rebooting the Web of Trust I. https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust/blob/master/final-documents/smart-signatures.pdf

Edström, Robert and Jack Pettersson. 2016. “Safer Smart Contracts through Type-Driven Development”. Chalmers University of Technology. http://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf.

Todd, Peter. 2016. “Closed Seal Sets and Truth Lists for Better Privacy and Censorship Resistance”. Peter Todd. https://petertodd.org/2016/closed-seal-sets-and-truth-lists-for-privacy.

Todd, Peter. 2016. “Building Blocks of the State Machine Approach to Consensus”. Peter Todd. https://petertodd.org/2016/state-machine-consensus-building-blocks

Todd, Peter. 2016. “Dex: Deterministic Predicate Expressions for Smarter Signatures”. Rebooting the Web of Trust II: ID2020 Workshop. https://github.com/WebOfTrustInfo/ID2020DesignWorkshop/blob/master/topics-and-advance-readings/DexPredicatesForSmarterSigs.md

Thomas, Stefan. 2016. "Crypto Conditions". Stefan Thomas. https://github.com/interledger/rfcs/tree/master/0002-crypto-conditions

Additional Credits

Authors: Christopher Allen, Shannon Appelcline

Contributors to Previous Papers: Greg Maxwell, Peter Todd, Ryan Shea, Pieter Wuille, Joseph Bonneau, Joseph Poon, and Tyler Close

About Rebooting the Web of Trust

This paper was produced as part of the **Rebooting the Web of Trust II** design workshop. On May 21st and May 22nd, 2016, over 40 tech visionaries came together in Manhattan, New York following the ID2020 Summit at the UN to talk about the future of decentralized trust on the internet with the goal of writing 3-5 white papers and specs. This is one of them.

Workshop Sponsors: Blockstack, Blockstream, Evernym, IPFS, Microsoft, Netki, Tierion, ID2020

Workshop Producer: Christopher Allen

Workshop Facilitators: Christopher Allen with graphic facilitation by Sue Shea, additional paper editorial & layout by Shannon Appelcline, and additional support by Kiara Robles.

What's Next?

The design workshop and this paper are just starting points for Rebooting the Web of Trust. If you have any comments, thoughts, or expansions on this paper, please post them to our GitHub issues page:

https://github.com/WebOfTrustInfo/ID2020DesignWorkshop/issues

The next Rebooting the Web of Trust design workshop is scheduled for October 19th-21st in San Francisco, California. If you'd like to be involved or would like to help sponsor these events, email:

ChristopherA@LifeWithAlacrity.com