Ledger Design Language: Designing and Deploying Formally Verified Public Ledgers

Nadim Kobeissi
INRIA Paris, Symbolic Software

Natalia Kulatova
INRIA Paris

***Abstract*—Cryptocurrencies have popularized public ledgers, known colloquially as "blockchains". While the Bitcoin blockchain is relatively simple to reason about as, effectively, a hash chain, more complex public ledgers are largely designed without any formalization of desired cryptographic properties such as authentication or integrity. These designs are then implemented without assurances against real-world bugs leading to little assurance with regards to practical, real-world security.**

Ledger Design Language (LDL) is a modeling language for describing public ledgers. The LDL compiler produces two outputs. The first output is a an applied-pi calculus symbolic model representing the public ledger as a protocol. Using ProVerif, the protocol can be played against an active attacker, whereupon we can query for block integrity, authenticity and other properties. The second output is a formally verified read/write API for interacting with the public ledger in the real world, written in the F⋆ programming language. F⋆ features such as dependent types allow us to validate a block on the public ledger, for example, by type-checking it so that its signing public key be a point on a curve. Using LDL's outputs, public ledger designers obtain automated assurances on the theoretical coherence and the real-world security of their designs with a single framework based on a single modeling language.

1. Introduction

Blockchain technology, pioneered by Bitcoin [1], provides a globally-consistent append-only ledger that does not rely on a central trusted authority. This underlying *public ledger* technology has led to the rise of more advanced cryptocurrencies with diverse and innovative use-cases. Ethereum [2], for example, deploys an entire state machine on top of a public ledger, which can then run programs called "smart contracts" in a globally shared namespace. Zerocash [3], on the other hand, focuses on making cryptocurrency trading and exchange more anonymous by employing zero-knowledge proofs. All of these technologies utilize the same underlying public ledger concept, where a series of append operations, each constituting a *block*, create a chain of blocks each cryptographically authenticating itself and the last block, hence the name "blockchain".

All of these public ledgers are based on decades-old concepts in computer science, namely binary hash chains and Merkle trees [4]. The Bitcoin blockchain is, in essence, an extended binary hash chain capable of supporting more complex data types and with certain rules regarding append and read operations. More complex designs such as Ripple [5], for example, uses a Merkle tree-shaped public ledger.

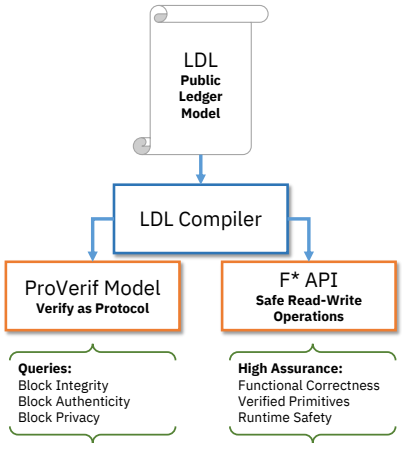

Figure 1. LDL Architecture Outline

1.1. A Language for Designing Public Ledgers

Given what we know about the concepts underlying all public ledgers, and the fact that they currently shoulder currencies of significant value, we propose Ledger Design Language (LDL), which combines two different aspects of formal verification in order to bolster the security of new public ledger designs in two distinct ways. LDL is composed of three components, illustrated in Figure 1:

1) **Input: A Modeling Language for Public Ledgers.** The LDL language describes public ledgers as a series of block types. As seen in Figure 3, each block type contains:
   - **Block Data Structure.** A listing of the elements this block contains. Each element can have a simple type, such as `string` or `integer`, or a type that carries security connotations, such as `signature` (a cryptographic signature which must be validated according to certain rules) or `nonce` (an identifier which must never be reused across the public ledger's existence.)
   - **Block Properties.** Two mandatory properties, `before` and `branchType`, describe the admitted directly preceding blocks, and whether it is followed by multiple branches (like a branching hash chain) or whether it

⟨*ldl*⟩ ::= ⟨*ledger*⟩ | ⟨*option*⟩

⟨*ledger*⟩ ::= 'ledger ' @identifier '{' (⟨*st*⟩)*'}'

⟨*option*⟩ ::= 'option ' ⟨*opt*⟩

⟨*opt*⟩ ::= 'vrf'

⟨*st*⟩ ::= ⟨*block*⟩ | ⟨*actor*⟩ | ⟨*struct*⟩

⟨*block*⟩ ::= 'block ' @identifier '[' (⟨*type*⟩ @identifier ';')* ']' '(' ('before' @identifier ';' 'branch' ⟨*branchType*⟩) ')'

⟨*actor*⟩ ::= 'actor ' @identifier '['']' '(' ('ownsChain' @identifier ';' 'ownership' ⟨*ownershipType*⟩) ')'

⟨*struct*⟩ ::= 'struct ' @identifier '[' (⟨*type*⟩ @identifier ';')* ']' '()'

⟨*type*⟩ ::= 'string' | 'address' | 'nonce' | 'integer' | 'signature' | 'date' | 'reference' | 'struct' | 'signpub'

⟨*branchType*⟩ ::= 'single' | 'multiple' | 'binary'

⟨*ownershipType*⟩ ::= 'single' | 'multiple'

Figure 2. LDL syntax.

is followed by a single branch (like a hash chain resembling the main Bitcoin blockchain). If the `binary` qualifier is used to describe the `branchType` of all block types on the ledger, then the LDL compiler types the ledger as a Merkle binary prefix tree. Once these two properties are described for each block, we can obtain a full map of the public ledger's structure.

2) **Output: A Symbolic Model Representation of the Public Ledger.** A model, written in the applied-pi calculus represents the public ledger as a protocol being played across a network with parallel processes. This model is adapted for analysis using the ProVerif [6] protocol verifier, which then analyzes it against an active attacker in the Dolev-Yao modeland attempts the validation of whether specific cryptographic properties, such as authenticity, are obtained.
3) **Output: An API Implementing the Public Ledger.** F⋆ [7] is an ML-like language aimed at program verification. Using its advanced features, such as dependent types and refinements, we can obtain a real-world API implementing the public ledger in a way that allows immediate use in production systems, while maintaining runtime safety, type safety and employing formally verified cryptographic primitives.

LDL's architecture aims at simplifying the design process of public ledgers, allowing designers to immediately obtain a way in which to rationalize regarding the effective security goals of their public ledger and to test it in a practical setting. Modifying the resulting ProVerif model allows live prototyping on a protocol logic level, while the F⋆ API allows for performance-related and other kinds of practical testing.

1.2. Related Work

Despite the unified and well-understood nature of the concepts underlying public ledgers, there appears to be, to the best of our knowledge, little to no prior work on applying formal verification to public ledgers. The Solid Ether project [8] targets Solidity, the language in which Ethereum "smart contracts" are written, and includes translation to F⋆ from both Solidity and its underlying bytecode which is what is stored on the Ethereum public ledger. Simplicity [9] follows another route by introducing an alternative to Solidity that lends itself more easily to writing correct programs. However, the structure of the ledger itself is not within the scope of these project's verification or correctness efforts.

2. The LDL Language

LDL prioritizes simplicity, aiming to construct a modeling language with the least amount of top-level components. In this section, we describe minimal LDL syntax and give a practical use-case example.

2.1. Language Semantics

LDL syntax (described in Figure 2) offers only three possible kinds of top-level declarations:

- **Block Declarations.** As demonstrated in Figure 3, `block` declarations allow the block types of a public ledger to be defined, first as a data structure and then with properties. The mandatory `before` and `branch` properties, specified for each block, are sufficient to obtain the full topology of the public ledger being described, be it a Merkle tree or a hash chain.
- **Actor Declarations.** Actors, defined using a minimal syntax (as demonstrated in Figure 4) represent "entities" which are specified to "own" a particular chain or branch that is part of the ledger. This is useful when translating to a symbolic protocol model and when generating the F⋆ API, as described in §3.
- **Structures.** A simple syntax exists for defining data structures, which can then be used as shorthand within blocks (for example, a `userInfo` structure can contain three strings describing a user's name, email and phone number.)

Block data types such as `signature` and `nonce` carry connotations of cryptographic security, which are translated in a special way by the LDL compiler.

2.2. LDL Features

LDL can be used to express some advanced features that do not usually appear in simpler public ledger designs (such as a ledger that tracks banking transactions). Nevertheless, these features do appear in more complex ledgers that are the basis for our case study in §4:

- **Verifiable Random Functions.** VRFs [10] are essentially a type of hash function with public key elements. The owner of an asymmetric key pair (sk, pk) for VRF can compute $VRF_{sk}(x) = (p, y)$, where x is an input value and y is a pseudorandom output value with uniform distribution. Meanwhile, p is a "proof value" that can be used by any public observer that possesses (pk, x, p, y) to verify that y is indeed a correct pseudorandom mapping of x under a VRF keyed with secret key sk. VRFs are important for many advanced public ledger designs:

they are used to encrypt all block contents by generating encryption keys that can be authenticated by the public ledger owner. In events where the public ledger is structured as a Merkle binary prefix tree, VRFs are used to randomize the indices to which blocks are allocated on the tree. The public ledger owner can then selectively reveal B_{i^p}, the proof value for any block B_i, allowing the correct index to be confirmed.

- **Cross-Referencing.** LDL also allows for *cross-referencing*, wherein blocks may include index references to other blocks on the chain.
- **Standard Cryptographic Functions.** In LDL, all encryption operations are translated into ProVerif and F* as operating under the AES-GCM AEAD cipher. All public key operations are translated as using Curve25519, and all signing operations use ED25519. This design ensures that safe primitives are used always, lessening the degree to which LDL model designers can make low-level cryptographic mistakes. We accept the expense this decision has on cipher suite flexibility.
- **Key Rotation.** Using the `signpub` datatype, a public ledger user can append a block that determines their identity's future signing public key for all blocks coming after this block, achieving a form of key rotation that is essential for implementing some of the public ledger designs discussed in §4.

2.3. Current Security Queries

We are currently able to query for the following basic security properties:

- **Block Integrity.** Any internal, non-leaf block delivered to the reader process must have been written as-is by the writer process.
- **Block Authenticity.** Any block delivered to the reader process must be cryptographically validated by a signature belonging to a certain actor if and only if it was sent by the writer process under that actor's identity.
- **Block Privacy.** Specifying `option vrf;` in the LDL model enables "VRF mode", in which the usage of a VRF is assumed for encryption across the entire ledger. In the event that the ledger describe is shaped as a Merkle binary prefix tree, the LDL compiler output will also assume the randomized, privacy-preserving distribution of indices. This latter property is important for public ledgers such as CONIKS (as discussed in §4.)
- **Reference Integrity.** A block on the ledger can contain references to other block indices. We want LDL output to model and give assurances on whether these pointers will always point to the correct blocks as intended. In some structures, this is impossible to guarantee: we want the compiler output to reflect this and to give useful insight to the designer.

2.4. Additional Work on Security Queries

In addition to the scope of this paper, we plan to include additional security queries which will allow us to reason on the security of the following events:

- **Chains with Shared Ownership.** Examples given in this paper show chains owned by a single actor. We want to expand our formalisms to include scenarios where Alice and Bob both contribute to the same chain, but with a specific logic governing which signatures are expected for which block. At this stage, inferences can be made based on block address references and hierarchy which allow LDL to determine which signatures to expect at which time.
- **Block Ordering Reliability.** Our current reasoning on the constructed ProVerif output assumes a perfectly ordered sequence of messages between the writer and accumulator processes. We want to expand this model to include security queries with regards to unreliable communication between these to processes, so that we can ask whether this can lead to invalid or unintended public ledger architectures being constructed.

3. From LDL Model to Formal Verification

In this paper, we give a high-level description of the reasoning and strategy behind LDL's outputs. Future work will discuss details such as translation rules and the logic of underlying security goals.

3.1. Verifying Public Ledgers as Symbolic Protocols

ProVerif models are composed of a set of process descriptions, functions, reductions and constructors in the symbolic model. In this model, all cryptographic primitives are perfect "black boxes". Then, a top-level process execution is described, which could include a parallel conjunction of processes. These processes send and receive information over a set of channels. On public channels, an active Dolev-Yao attacker can monitor and tamper with messages. In this context, events and queries are defined with respect to the exposure (or lack thereof) of certain secret variables.

An LDL model is translated into this setting by specifying three main processes:

- **Writer Process.** This process issues the blocks in the order mandated by the structure of the public ledger, using the identities defined by `actor` declarations.
- **Accumulator Process.** This process accumulates the blocks as received over the network by the writer process, producing a structure that acts as the record for the public ledger.
- **Reader Process.** This process then reads blocks from the ledger. Read operations are sent to the accumulator process, which then delivers block information over the network to the reader process. This allows ProVerif a chance to expose this block information to the active attacker.

3.2. Generating a Verified Public Ledger API in F*

In F*, we want to obtain an API implementing the public ledger in a way that allows for safe, high-assurance read-write operations and with a formally verified cryptographic context. We expose an API with top-level access functions generated through LDL `actor` declarations:

```
block Root[
       date currentDate;
       nonce merkleRoot;
       string ledgerName;
](
       before UserChain;
       branch multiple;
)
block UserChain[
       string name;
       signpub initialKey;
](
       before Claim;
       branch single;
)
block Claim[
       date timestap;
       struct claimData;
       signpub nextKey;
](
       before Claim;
       branch single;
)
```

Figure 3. A partial LDL model showing two block declarations. A `UserChain` describes the root block of an append-only Merkle branch and indicates that it can occur only before the `Root` block for the entire public ledger. Meanwhile, the `Claim` describes a check which can only have a single, non-branching chain of `Claim` blocks as children. The full model describes the ledger discussed in §4.

3.2.1. Blocks as Dependent Types

Using dependent types in F*, we can describe blocks as a collection of simple datatypes and also algebraically validated types. This allows us to type-check a block as valid within the context of a valid signature (which authenticates to a particular identity and is well-formed as two points on the curve, in the case of Ed25519) and a hash that is coherent within the hash chain of the branch in which it belongs.

3.2.2. Functional Correctness and Runtime Safety

The F* API will also be able to guarantee that all operations on the public ledger target a ledger that is well-formed, according to spec, with no room, for example, for misdirected block references.

3.2.3. Verified Primitives with HACL*

HACL* [11] is a formally verified cryptographic library in F*. Primitives within HACL* are verified for functional correctness, memory safety and against side-channel attacks. These guarantees carry when HACL* is translated to C code. The LDL compiler calls HACL* primitives for all cryptographic operations and thereby benefits from these security guarantees.

4. Case Study: an LDLChain Implementation

We present a case study that uses LDL to implement and verify "LDLChain", a full public ledger design that can be used for managing public key information. In order to understand how to best shape LDLChain, we based ourselves on an extensive literature review of existing public ledger systems. LDLChain is based on CONIKS [12], a public ledger approach to certificate transparency by Melara, Blankstein, Bonneau, Felten and Freedman. LDLChain is also is inspired by ClaimChain [13], a recent design by Kulynych, Isaakidis, Troncoso and Danezis, which proposes a public ledger that lends itself towards storing claims made by users regarding their long-term identity keys for various cryptographic systems that employ public-key encryption.

```
actor ChainOwner[] (
       ownsChain Root;
       ownership single;
)
actor Alice[](
       ownsChain UserChain;
       ownership single;
)
```

Figure 4. LDL actors that are part of LDLChain, discussed in §4.

```
type claimData = |InitClaimData:
       screen_name: option string →
       real_name: option string →
       identifiers: list indentifier →
       keys: list keyEnt →
       hashMetadata: bytes → claimData
type claim = |InitClaim:
       timeStamp : (k : time { k> 0}) →
       claimData: claimData →
       signpub: bytes →
       signature : bytes {
              verify signature (
                     concat (toBytes claimData) (
                            toBytes key
                     )
              )} → claim
```

Figure 5. Target F* code representing the datatypes generated from Figure 3 by the LDL compiler. Signatures are dependently typed so as to be verified as valid and authentic.

Given the space constraints within this paper, we have found it impossible to describe our case study implementation in detail, including vital information such as the full LDL model, its translation into the applied-pi calculus and its F* API. A longer version of the paper will contain a complete version of the case study.

4.1. LDLChain Design and Security Goals

LDLChain blocks (described in Figure 3) can contain self-identifying information, such as usernames, and endorsements of claims made by other users (by referencing the block index on which they were made). LDLChain blocks can also contain key information: LDL allows for key rotation using the `signpub` datatype.

LDLChain is largely based on ClaimChain, and its security properties are exactly that of ClaimChain.[1]

- **Authenticity.** The information stored in a ClaimChain has been input by the owner of the chain.
- **Integrity.** The information stored in a ClaimChain has not been modified since it was added to the chain.
- **Privacy.** Only readers authorized by the ClaimChain owner to access a claim at a particular point in time can read the content of that claim, at that point in time.
- **Non-equivocation.** At a particular point in time, a ClaimChain owner cannot provide two authorized readers of a claim with different content for that claim.

1. The main reason why we implement LDLChain instead of simply ClaimChain is that, to the best of our knowledge, ClaimChain's public ledger data structure is not currently fully defined by its authors.

These properties can be described in LDL, with the exception of non-equivocation, which is currently slated for future work.

4.2. The LDLChain API in F*

We briefly describe LDLChain's F* API.[2] LDL's F* library comes with implementations for standard data structures such as Merkle trees and Skip lists. Depending on the LDL model, these libraries are used in the generation of an F* `Spec`. In it, blocks are represented as complex types with dependent subtypes depending on the structures described in LDL, as seen in Figure 5. For that F* type, an API is generated, including, for example, a function `addClaim`, which uses lemmas and logic contained in the LDL Merkle tree F* library in order to manage the secure insertion of the claim into the public ledger.

5. Conclusion

In this paper, we introduce LDL as a domain-specific modeling language that can be used to describe public ledgers. The LDL compiler validates these descriptions and generates two distinct outputs that allow users to reason about the security of public ledgers, obtain meaningful conclusions and deploy them with practical, verified implementations.

5.1. Future Work

Future work includes extending the LDL syntax to capture more public ledger use-cases and continuously improving the language's functionality and expressiveness in accordance with modern public ledger design norms.

Currently, there exists no formal proof guaranteeing that the LDL translation rules fully preserve all security goals that are targeted by an LDL model as it is translated to an applied pi calculus model or to F* code. Future work aims to remedy this.

Furthermore, different avenues could be explored for LDL output. The CryptoVerif [14] verifier accepts a similar syntax to ProVerif, but can be used to describe computational models with algebraic properties. LDL's F* translations themselves could also be expanded to capture more of the protocol logic which we currently verify using ProVerif, creating a formal relationship between the two representations.

5.2. Acknowledgments

We thank Karthikeyan Bhargavan, Bruno Blanchet and Antoine Delignat-Lavaud for discussions regarding translation to ProVerif and F*. We also thank Hadi Kabalan and Russell O'Connor for discussions regarding use-cases. We also acknowledge Diffie the poodle for his profound insight and moral support.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," http://gavwood.com/paper.pdf.

[3] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 459–474.

[4] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1987, pp. 369–378.

[5] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

[6] B. Blanchet, "Automatic verification of security protocols in the symbolic model: The verifier proverif," in *Foundations of Security Analysis and Design VII*. Springer, 2014, pp. 54–87.

[7] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*. ACM, 2016, pp. 256–270.

[8] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin, "Formal verification of smart contracts," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS16*, 2016, pp. 91–96.

[9] R. O'Connor, "Simplicity: A new language for blockchains," *CoRR*, vol. abs/1711.03028, 2017. [Online]. Available: http://arxiv.org/abs/1711.03028

[10] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 120–130.

[11] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[12] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users." in *USENIX Security Symposium*, 2015, pp. 383–398.

[13] B. Kulynych, M. Isaakidis, C. Troncoso, and G. Danezis, "Claimchain: Decentralized public key infrastructure," *CoRR*, vol. abs/1707.06279, 2017. [Online]. Available: http://arxiv.org/abs/1707.06279

[14] B. Blanchet, "Cryptoverif: Computationally sound mechanized prover for cryptographic protocols," in *Dagstuhl seminar on Applied Formal Protocol Verification*, 2007, p. 117.

2. Due to space constraints, we cannot include a similar description of the ProVerif output in this version of the paper.