Formal Specification and Verification of Hyperledger Fabric Chaincode

Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl
Institute of Theoretical Informatics
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
beckert@kit.edu, herda@kit.edu, kirsten@kit.edu, jonas.schiffl@student.kit.edu

***Abstract*—Smart contracts are programs building on blockchain technology. They implement functionality that has been agreed on between the concerned parties on a network. However, their immutability and exposed position make them vulnerable to programming errors, leading to faulty behavior and possible exploits. Therefore, smart contracts demand a particularly thorough analysis, ideally using formal program verification. In this paper, we present an approach for the deductive verification of Hyperledger Fabric smart contracts using the KeY prover. We have extended KeY to handle Fabric ledger implementations; in particular, we have developed mechanisms for reasoning about serialization and object persistence. The feasibility of our approach is demonstrated with a small case study.**

***Index Terms*—formal program verification, blockchain, smart contract, Hyperledger Fabric.**

Acknowledgments

The work presented here is part of the project *VeriSmart – Specification and Verification of Smart Contracts* within the BMBF-funded programme Software Campus.

I. Introduction

Distributed ledger technology and blockchains are rapidly gaining both popularity and maturity for business applications. Blockchain systems enable the establishment of trade networks, financial services, and many other systems that depend on consistency and trust even when the concerned parties do not fully trust each other.

Smart contracts take blockchain technology to yet another level. While a blockchain transforms private data into shared data, i.e., a distributed ledger, smart contracts transform private programs into shared programs, providing a transparent and regulated interface to access and update this ledger [1].

Smart contracts have already garnered lots of attention with the rise of, most prominently, Bitcoin [2] and Ethereum [3], but they are present in numerous other systems as well, e.g., Stellar (www.stellar.org), Ripple (www.ripple.com), or RootStock (www.rsk.co). What all smart contracts have in common is that they manage value, in the form of digital currencies or tokens representing real-world assets. Thus, faulty program behavior can immediately cause losses for the contract parties. Furthermore, smart contracts cannot be easily changed, since this requires the consent of all concerned parties. In a public blockchain setting, smart contracts are entirely public, and even in a private blockchain network, network-wide visibility of contracts can be useful. Therefore, smart contracts also are interesting targets for attackers who examine the program's code to find exploitable bugs. We, hence, argue that smart contracts must be thoroughly checked and analyzed, ideally using formal verification methods. Then, both the developers and the users of contracts can be provided with a trusted mathematical correctness proof as a guarantee that the contract behaves as specified.

In this work, we present an approach for the formal specification and verification of smart contracts for Hyperledger Fabric (see Section II) written in Java. We use KeY (see Section III), a system for proving the formal correctness of annotated Java source code, as our verification platform. We extend KeY to integrate the Hyperledger Fabric API and to enable reasoning about serialized objects that are stored on the ledger (Section IV). In Section V, we demonstrate the feasibility of our approach using a small case study: We present an implementation of the Rock-Paper-Scissors game as a Fabric smart contract, alongside with formal specifications, which we verify using KeY. Reasoning about serialized objects still requires some interaction, but the contract's functional properties are proven fully automatically.

II. Hyperledger Fabric

Hyperledger [4] is a collaboration aiming to develop blockchain technologies for business use cases. Initiated in 2015, it is hosted by the Linux Foundation and supported by numerous companies from the technology and financial services industries. Initially contributed by IBM and Digital Asset, Hyperledger Fabric (often shortened to *Fabric*) was among the first projects emerging from Hyperledger in 2016.

Fabric focuses on private, permission-based blockchains, i.e., network participants are preselected by a central authority, and smart contract function calls can be limited to a subset of network participants. Fabric does not have a built-in cryptocurrency, although one can be implemented if needed. In a Fabric network, the participants (called *peers*) form communication channels. For each channel, there is a ledger, which is stored on every peer in a consistent manner. Smart contracts, called *chaincode* in the Fabric context, are programs, installed on the peers, that provide controlled access to the ledger. Furthermore, they define the functionality that

has been agreed on, and the assets that are stored on the ledger. Chaincode can be written in Go or Java; future versions of Fabric are meant to support more languages.

In the Fabric context, a transaction is an invocation of a smart contract function from an application or a client outside of the network. Each such invocation is performed locally on the peer and atomically with respect to the ledger as only after code execution, the peer creates a transaction from the performed operations and passes it to the ordering service. The transaction first needs to be accepted by one or more peers (as defined in the chaincode's endorsement policy) before a client transfers the transaction to the ordering service, where the transactions are arranged in blocks and redistributed to the peer nodes who then update the ledger accordingly. Since Fabric is private and permission-based by default, there is no need for a proof-of-work consensus algorithm, resulting in very high transaction speed compared to the Bitcoin or Ethereum networks [5].

Our approach focuses on statically (i.e., before actually executing the code) verifying functional correctness, non-exceptional behavior, and termination of the chaincode, so that the caller of a function as well as the peers – where the function is executed – and the author of the code can be sure that the code behaves as specified.

III. The KeY Verification System

KeY [6] is a semi-interactive deductive theorem prover for statically verifying sequential Java programs formally specified using the Java Modeling Language (JML) [7]. In JML, each Java method can be annotated with pre- and postconditions, together with the program locations which the method's execution may change, such that verification can be done modularly. A successful verification using KeY proves that the method, when started in a state satisfying the precondition, terminates in a state satisfying the postcondition, and that it only affects those memory locations for which changes are allowed by the specification. Modularity is done at method level, i.e., method calls can be handled in proofs by replacing them with the logical translation of their JML contracts. Specification in JML allows using side-effect-free Java expressions in first-order predicates with extensions such as, e.g., quantifiers and abstract data types.

Verification is done by automatically translating the specified Java method into a – logically equivalent – formula in Java dynamic logic (JavaDL) [8], where the source code can be executed and reasoned about inside the logic using symbolic execution to capture the possible effects the program method in logical formulas.

KeY uses a sequent calculus, consisting of various deduction rules, which may either be applied automatically or interactively. As verification with KeY is done statically, i.e., without actually running the code, a successful verification produces a formal proof which universally guarantees that, whenever the verified code is used in a setting where the specified precondition is satisfied, the program will satisfy the specified postcondition and change at most the specified memorys locations. An advantage of static analysis is that no runtime overhead is incurred as is the case with other approaches such as runtime monitoring.

KeY allows, moreover, to define custom extensions to the JML language such as theories on abstract data types. Two prominent examples are the theory of the String data type and the theory of finite sequences [9]. Finite sequences are commonly used as abstractions of Java arrays, but also for other finite ordered structure. KeY allows writing definitions of such theories as well as deduction rules via so-called *taclets* [10]. Taclets are lightweight entities for logical and technical rule definitions, which can be easily extended and implemented by the user. New taclets, which extend the calculus, need to be proven, i.e., shown to be deducible from the existing taclets. Proofs can often be constructed automatically by the built-in strategies based on configurable heuristics and various predefined macros.

IV. Approach

In this section, we describe how we have extended the KeY system to support verification of functional properties for smart contracts written in Java for the Hyperledger Fabric framework – in addition to the verification of regular sequential Java programs. Note that verifying the correctness of the Fabric framework itself (e.g., communication between peers and orderer), on which the contracts are executed, is not within the scope of this work. The fact that individual invocations of smart contracts are atomic justifies their verification as sequential Java programs. In this work, we focus on the verification of functional properties of single smart contract functions. Nevertheless, our approach can be extended to verifying properties regarding function call sequences.

What distinguishes smart contracts from other Java programs are read and write operations on the distributed ledger. In order to support the verification of these operations in KeY, we add JML specifications of the Fabric API methods for reading and writing data on the ledger to KeY. We assume that the implementation of the API methods fulfills these specifications, i.e., the verification within this work targets the *user-specific* part and not the API methods as they are part of the general fabric infrastructure.

A second distinguishing feature of Hyperledger smart contracts is the use of object serialization resp. deserialization when writing and reading data to and from the ledger. In this section, we formulate guarantees that serialization and deserialization need to fulfill and that, hence, can be assumed after the usage of such operations in smart contracts. As this problem can be dealt with and expressed separately to that of verifying smart contracts, we do not verify the implementation of the (de)serialization functions in this work. Instead, we make use of the general assumption that the provided deserialization function is the inverse of the provided serialization function. The user must provide their own or find some general implementation of the (de)serialization functions that fulfills these guarantees.

In what follows, we explain how we extend the logic of the KeY theorem prover with formalizations to deal with both features – reading/writing and (de)serialization – and enable the verification of smart contracts under these guarantees.

A. Specification of the Hyperledger Fabric API Methods

The Fabric API class `ChaincodeStub` provides the functionality for reading, writing, and deleting data from the ledger, using the methods `getState`, `putState`, and `deleteState`, respectively. We provide specifications for these three methods that can be used when proving smart contracts. For modelling the ledger itself, we make use of a JML extension provided by KeY to reason about the theory of finite sequences (an abstract data type as described in Section III). We model the distributed ledger as a sequence of entries, where each entry has an id of type Integer[1] and a value expressed as a sequence of bytes. We specify the methods `getState`, `putState` and `deleteState` as follows:

- `getState` takes an id and returns the last entry (which may be the special constant `deleted`, see below) in the sequence with this id. If no such entry exists, the special constant `empty` is returned.
- `putState` takes an entry (with an id and a value) and adds it to the end of the sequence.
- `deleteState` takes an id and adds a new entry to the sequence with the given id and a special constant `deleted`.

The specification of the API methods enables the verification of smart contracts.

B. Serialization and Deserialization

In a typical Fabric smart contract written in Java, heap-based data structures (consisting of objects) are serialized and the resulting byte sequences are written to the ledger, or, conversely, byte sequences are read from the ledger and used to reconstruct a structure of Java objects. This poses a challenge for verification: we need to show that a structure that was serialized and written to the ledger is *equal* – in some sense – to one that was read and deserialized from the ledger, even though the two objects have different object identities.

We define two objects (and the structures they refer to) to be equal if and only if all their respective fields are equal (note that this is a recursive definition). We argue that for the use in smart contracts, the objects that are serialized and written to the ledger mostly serve as data records consisting of several primitive data types. If the pointer structure of the data read and written on the ledger is relevant to the application, then a stricter definition of equality may be used, adding an axiom to express that serialization and deserialization preserve isomorphism of the pointer structure (and not just values).

We define an equality relation between objects using a new abstract data type (see Section III) for each object type that is written to the ledger. Given a Java class C, we define an abstract sort S_C and, for each field f of type T, we define function $getF : S_C \to T$. We also define a constructor $newC : (T_1 \times \cdots \times T_n) \to S_C$ that takes the values for all fields $f_1 : T_1, \ldots, f_n : T_n$ of the class C and returns an abstract object. For every field, we add a rule stating that the function $getF$ for a field f, when called on a constructor, returns the value of the parameter corresponding to f. This allows us to formalize equality the following way: two values of type S_C are equal iff the value returned by *getF* for every field f of C is equal. We are now able to specify and verify the equality of objects that are read and written to the ledger.

Furthermore, we define a function γ_C for transforming a Java object of class C and a heap to an abstract data type S_C in the following way: $\gamma_C : C \times H \to S_C$ (where H is the set of all heaps). This function allows us to make the connection between Java objects and their abstract representations. We add a calculus rule for this function using the constructor function. Thus, $\gamma_C(o, f)$ equals $newC(select(h, o, f_1), \ldots, select(h, o, f_n))$.

Finally, we define two JavaDL functions, $serializeC : S_C \to Seq$ and $deserializeC : Seq \to S_C$, that represent object serialization and deserialization, respectively. We are now able to logically express that deserializing a serialized value always returns the value previously written to the ledger (we use the equality rule of abstract data types), which we will use to specify the serialization and deserialization methods in Java that are used in the smart contracts.

All functions and rules mentioned above are defined in a generic way and may thus be generated automatically by KeY, given the Java class of the smart contracts, which is a natural extension to the work presented here.

V. Example

We explain the concepts introduced in the previous section using a Hyperledger Fabric implementation of the *Rock-Paper-Scissors* game. The class `Game` shown in Listing 1 represents instances of the game. There are two fields, `sign1` and `sign2`, where the first and second player, respectively, can set their sign, as well as a field `winner` where the winner of the game is stored once a game is finished. Instances of this class are stored on the ledger. Both methods can be easily specified using the JML extensions described in Section IV.

In the following, the JavaDL sort *kGame* represents the abstract type of a `Game` object, together with the functions for each of its fields `sign1`, `sign2` and `winner`.

```
public class Game {
  int sign1, sign2, winner;
  static abstract byte[] serialize(Game g);
  static abstract Game deserialize(byte[] bytes);
}
```

Listing 1. The Java implementation of the Game type.

We exemplarily present the smart contract function which determines the winner of a rock-paper-scissors game in Listing 2. The methods `writeGame` and `readGame` are short hands for serializing and storing, and reading and deserializing objects to and from the ledger, respectively. The method

[1]However, the actual API uses Strings instead. Within this paper, we changed it for the sake of simplicity.

sign1Won takes two integers representing the rock, paper or scissors sign (represented by the corresponding constant Integer fields) of both players as parameters (i.e., the first parameter holds the first player's sign, etc.) and returns true iff the first sign beats the second sign, i.e., iff the first player wins. The method getWinner reads a game from the ledger and writes 1 in the field winner iff the first player has won, 2 iff the second player has won or 0 if there is a draw. The specification of getWinner states that, if sign1 beats sign2 when calling getWinner, then, after the call of getWinner, the last entry of the game with the id gameId in the ledger has 1 written in the winner field. This can easily be specified in JML using our extensions and the auxiliary method sign1Won, also shown in Listing 2 (which captures the core game semantics of which sign beats which other sign). The JML keyword \result specifies the return value of a method, which can be used in postconditions (the keyword ensures indicates the postcondition in JML). The term after assignable describes the locations to be changed at most during execution of the method (the keyword \nothing denotes the empty location set). We have verified the method getWinner as well as methods for creating a new game, setting a sign, the two methods writeGame and readGame and other auxiliary methods. Using our extended version of KeY, verification was mostly automatic, with a couple of interactive rule applications (which should be easy to automate in future versions).

```
public class RPS extends ChaincodeBase {
  public Response getWinner(ChaincodeStub stub,
                            int gameId) {
    Game game = readGame(stub, gameId);
    int winner = 0;
    int s1 = game.sign1; int s2 = game.sign2;
    if (sign1Won(s1, s2)) winner = 1;
    else if (sign1Won(s2, s1)) winner = 2;
    game.winner = winner;
    writeGameToLedger(stub, gameId, game);
    return newSuccessResponse();
  }
  ...
  /*@ ensures \result <==>
    @  (s1 == ROCK && s2 == SCISSORS)
    @  || (s1 == PAPER && s2 == ROCK)
    @  || (s1 == SCISSORS && s2 == PAPER);
    @ assignable \nothing;
    @*/
  private boolean sign1Won(int s1, int s2) {
    return (3 + s1 - s2) % 3 == 1;
  }
}
```

Listing 2. The smart contract for getWinner.

VI. Related Work

Despite the recency of the rise of smart contracts, there have been some attempts at formal verification specifically targeted at this new technology. Due to the popularity of Ethereum, these usually target either Ethereum's Solidity language or EVM, the associated bytecode. A number of these works have already been systematically overviewed [11]. One approach is to check contracts for common (anti-)patterns identified in smart contracts vulnerable to exploits [12]–[14]. Another verification approach is to write smart contracts in a language that is more suited to formal verification such as, e.g., F* [15], and to provide a canonical translation to Solidity or EVM. One such solution was temporarily built into Remix (remix.ethereum.org), a tool for writing, testing and deploying Solidity smart contracts, enabling users to formally specify smart contracts and prove their correctness using the deductive theorem prover Why3 [16]. Besides static verification, which is done before actually executing the code, there is also work on designing new blockchain programming languages which allow enconding security policies which are enforced at runtime [17]. Stepping away from imperative programming, there is research on writing declarative smart contracts that are closer to legal documents [15].

There exists, moreover, work on provably correct (general) distributed systems, which oftentimes, due to their complexity, targets only simplified distributed protocols. However, some approaches target implementations of complex distributed systems [18], [19]. They rely on formal axioms about (de)serialization functions, for which a formally verified library was developed in a student research project [20]. As for Hyperledger Fabric, there is ChaincodeScanner (chaincode.chainsecurity.com), a tool that checks Fabric smart contracts written in Go for anti-patterns, such as concurrency, dependence on global state or non-determinism. To the best of our knowledge, this paper is the first work on formal deductive verification of smart contracts for Hyperledger Fabric.

VII. Conclusion and Future Work

In this paper, we presented an approach for the formal verification of smart contracts for the Hyperledger Fabric framework. We have extended the KeY verification system such that it can handle the Fabric API methods for reading and writing on the ledger, and also deal with object (de)serialization. In a small case study, we validated our approach on a Fabric smart contract of the Rock-Paper-Scissors game written in Java. Specifying the ledger objects in a way that enables reasoning about reading and writing objects in KeY is a manual task as of yet, but can be automated. Proof construction is mostly automatic and only requires some interaction in the parts that deal with object serialization. We were able to automatically prove correctness of the game logic implementation. Generalizing from here, we expect that, with our approach, typical smart contracts can be easily verified once a formal specification is available.

As a next step, we plan to extend KeY to support the API methods for handling the transaction history of a ledger. In order to achieve this, methods for extracting and iterating over the history of an entry must be created in KeY. Moreover, we want to extend our approach to the verification of chaincode protocols, i.e., the interplay of several chaincode calls, which can be integrated using rely/guarantee-style reasoning [21]. Another future research question is how to integrate Hyperledger's concept of identity in order to reason about the permissions of a channel.

References

[1] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[2] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://www.bitcoin.org/bitcoin.pdf

[3] V. Buterin. (2013) Ethereum: A next-generation smart contract and decentralized application platform. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A distributed operating system for permissioned blockchains," in *13th EuroSys Conference (EuroSys 2018)*, R. Oliveira, P. Felber, and Y. C. Hu, Eds. ACM, 2018, pp. 30:1–30:15.

[5] P. Thakkar, S. Nathan, and B. Vishwanathan, "Performance benchmarking and optimizing Hyperledger Fabric blockchain platform," *CoRR*, vol. abs/1805.11390, 2018.

[6] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification - The KeY Book: From Theory to Practice*, ser. Lecture Notes in Computer Science. Springer, 2016, vol. 10001.

[7] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[8] B. Beckert, V. Klebanov, and B. Weiß, "Dynamic Logic for Java," in *Deductive Software Verification - The KeY Book: From Theory to Practice*, ser. Lecture Notes in Computer Science, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds. Springer, 2016, vol. 10001, ch. 3, pp. 49–106.

[9] P. H. Schmitt and R. Bubel, "Theories," in *Deductive Software Verification - The KeY Book: From Theory to Practice*, ser. Lecture Notes in Computer Science, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds. Springer, 2016, vol. 10001, ch. 5, pp. 149–166.

[10] P. Rümmer and M. Ulbrich, "Proof search with taclets," in *Deductive Software Verification - The KeY Book: From Theory to Practice*, ser. Lecture Notes in Computer Science, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds. Springer, 2016, vol. 10001, ch. 4, pp. 107–147.

[11] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *30th International Conference on Computer Aided Verification (CAV 2018)*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 51–78.

[12] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *2016 SIGSAC Conference on Computer and Communications Security (CCS 2016)*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269.

[13] J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, Eds., *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab*, ser. Lecture Notes in Computer Science, vol. 9604. Springer, 2016.

[14] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *2018 SIGSAC Conference on Computer and Communications Security (CCS 2018)*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 67–82.

[15] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor, "Evaluation of logic-based smart contracts for blockchain systems," in *10th International Symposium on Rule Technologies (RuleML). Research, Tools, and Applications*, ser. Lecture Notes in Computer Science, J. Alferes, L. Bertossi, G. Governatori, P. Fodor, and D. Roman, Eds., vol. 9718. Springer, 2016, pp. 167–183.

[16] J. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *22nd European Symposium on Programming (ESOP '13)*, ser. Lecture Notes in Computer Science, M. Felleisen and P. Gardner, Eds., vol. 7792. Springer, 2013, pp. 125–128.

[17] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in flint," in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, S. Marr and J. B. Sartor, Eds. ACM, 2018, pp. 218–219.

[18] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *36th SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, D. Grove and S. Blackburn, Eds. ACM, 2015, pp. 357–368.

[19] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill, "IronFleet: proving practical distributed systems correct," in *25th Symposium on Operating Systems Principles (SOSP 2015)*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 1–17.

[20] K. Simmons, "Cheerios," Student Research Project, 2016. [Online]. Available: https://courses.cs.washington.edu/courses/cse599w/16sp/projects/cheerios.pdf

[21] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, no. 4, pp. 319–340, 1976.