Abstract

Decentralized applications (dApps) consist of smart contracts that run on blockchains and clients that model collaborating parties. dApps are used to model financial and legal business functionality. Today, contracts and clients are written as separate programs – in different programming languages – communicating via send and receive operations. This makes distributed program flow awkward to express and reason about, increasing the potential for mismatches in the client-contract interface, which can be exploited by malicious clients, potentially leading to huge financial losses. In this paper, we present Prisma, a language for tierless decentralized applications, where the contract and its clients are defined in one unit. Pairs of send and receive actions that “belong together” are encapsulated into a single direct-style operation, which is executed differently by sending and receiving parties. This enables expressing distributed program flow via standard control flow and renders mismatching communication impossible. We prove formally that our compiler preserves program behavior in presence of an attacker controlling the client code. We systematically compare Prisma with mainstream and advanced programming models for dApps and provide empirical evidence for its expressiveness and performance.

The design space of dApp programming and other multi-party languages depends on one major choice: a local model versus a global model. In a local model, parties are defined in separate programs and their interactions are encoded via send and receive effects. In a global language, parties are defined within one shared program and interactions are encoded via combined send-and-receive operations with no effects visible to the outside world. The global model is followed by tierless [19, 9, 4, 5, 11, 25, 26, 20, 27] and choreographic [13, 16, 12] languages. However, known approaches to dApp programming follow the local model, thus rely on explicitly specifying the client-contract interaction protocol. Moreover, the contract and clients are implemented in different languages, hence, developers have to master two technology stacks. The dominating approach in industry is Solidity [15] for the contract and JavaScript for clients. Solidity relies on expressing the protocol using assertions in the contract code, which are checked at run time [1]. Failing to insert the correct assertions may give parties illegal access to monetary values to the detriment of others [17, 14]. In research, contract languages [10, 6, 23, 24, 8, 7, 18, 3] have been proposed that rely on advanced type systems such as session types, type states, and linear types. The global model has not been explored for dApp programming. This is unfortunate given the potential to get by with a standard typing discipline and to avoid intricacies and potential mismatches of a two-language stack. Our work fills this gap by proposing Prisma – the first language that features a global programming model for Ethereum dApps. While we focus on the Ethereum blockchain, we believe our techniques
to be applicable to other smart contract platforms. Prisma enables interleaving contract and client logic within the same program and adopts a direct style (DS) notation for encoding send-and-receive operations (with our \texttt{awaitCl} language construct) akin to languages with async/await [2, 22]. DS addresses shortcomings with the currently dominant encoding of the protocol’s finite state machines (FSM) [15, 6, 23, 24, 8, 7]. We argue writing FSM style corresponds to a control-flow graph of basic blocks, which is low-level and more suited to be written by a compiler than by a human. With FSM style, the contract is a passive entity whose execution is driven by clients. whereas the DS encoding allows the contract to actively ask clients for input, fitting dApp execution where a dominant contract controls execution and diverts control to other parties when their input is needed.

In the following Prisma snippet, the \texttt{payout} function is a function invoked by the contract when it is time to pay money to a client. In Prisma, variables, methods and classes are separated into two namespaces, one for the contract and one for the clients. The \texttt{payout} method is located on the contract via the annotation \texttt{@co}. The body of the method diverts the control to the client using \texttt{awaitCl(...)} \{ ... \}, hence the contained \texttt{readLine} call is executed on the client. Note that no explicit send/receive operations are needed but the communication protocol is expressed through the program control flow. Only after the check \texttt{client == toBePayed} that the correct client replied, the current contract balance \texttt{balance()} is transferred to the client via \texttt{transfer}.

```
@co def payout(toBePayed: Arr[Address]): Unit = {
  awaitCl(client => client == toBePayed) {
    readline("Press enter for payout")
  }
  toBePayed.transfer(balance())
}
```

Overall, Prisma relieves the developer from the responsibility of correctly managing distributed, asynchronous program flows and the heterogeneous technology stack. Instead, the burden is put on the compiler, which distributes the program flow by means of selective continuation-passing-style (CPS) translation and defunctionalisation and inserts guards against malicious client interactions.

We needed to develop a CPS translation for the code that runs on the Ethereum Virtual Machine (EVM) since the EVM has no built-in support for concurrency primitives which could be used for asynchronous communication. While CPS translations are well-known, we cannot use them out-of-the-box because the control flow is interwoven with distribution in our case. A CPS translation that does not take distribution into account would allow malicious clients to force the contract to deviate from the intended control flow by sending a spoofed continuation. Thus, it was imperative to prove correctness of our distributed CPS translation to ensure control-flow integrity of the contract.

2012 ACM Subject Classification Software and its engineering → Distributed programming languages; Software and its engineering → Domain specific languages; Software and its engineering → Compilers

Keywords and phrases Domain Specific Languages, Smart Contracts, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.35


Supplementary Material Software: https://github.com/stg-tud/prisma

Funding This work has been funded by the German Federal Ministry of Education and Research iBlockchain project (BMBF No. 16KIS0902), by the German Research Foundation (DFG, SFB 1119 CROSSING Project), by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, by the Hessian LOEWE initiative (emergenCITY), by the Swiss National Science Foundation (SNSF, No. 200429), and by the University of St. Gallen (IPF, No. 1031569).
References


Prisma: A Tierless Language for Enforcing Protocols


