Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications

DAVID RICHTER and DAVID KRETZLER, Technical University of Darmstadt, Germany
PASCAL WEISENBURGER and GUIDO SALVANESCHI, University of St. Gallen, Switzerland
SEBASTIAN FAUST and MIRA MEZINI, Technical University of Darmstadt, Germany

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 and 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.

CCS Concepts: • **Software and its engineering → Distributed programming languages**; Domain specific languages; Compilers.

Additional Key Words and Phrases: Domain Specific Languages, Smart Contracts, Scala

ACM Reference Format:
David Richter, David Kretzler, Pascal Weisenburger, Guido Salvaneschi, Sebastian Faust, and Mira Mezini. 2023. Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 17 (September 2023), 44 pages. https://doi.org/10.1145/3604629

1 INTRODUCTION

dApps enable multiple parties sharing state to jointly execute functionality according to a predefined agreement. This predefined agreement is called a *smart contract* and regulates the interaction between the dApp's clients. Such client–contract interactions can be logically described by state machines [55, 56, 79, 84] specifying which party is allowed to do what and when.

dApps can operate without centralized trusted intermediaries by relying on a blockchain and its consensus protocol. To this end, a contract is deployed to and executed on the blockchain, which guarantees its correct execution; clients that run outside of the blockchain can interact with the contract via transactions. A key feature of dApps is that they can directly link application logic with transfer of monetary assets. This enables a wide range of correctness/security-sensitive business

Authors' addresses: David Richter, david.richter@tu-darmstadt.de; David Kretzler, david.kretzler@tu-darmstadt.de, Technical University of Darmstadt, Hochschulstr. 10, 2052, 64289, Germany; Pascal Weisenburger, pascal.weisenburger@unisg.ch; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, Torstrasse 25, 9000, St. Gallen, Switzerland; Sebastian Faust, sebastian.faust@tu-darmstadt.de; Mira Mezini, mezini@informatik.tu-darmstadt.de, Technical University of Darmstadt, Pankratiusstraße 2, 2052, 64289, Germany.

2023. 0164-0925/2023/9-ART17 $15.00
https://doi.org/10.1145/3604629

applications, e.g., for cryptocurrencies, crowdfunding, and public offerings,[1] and the same feature makes them an attractive target for attackers. The attack surface is wide since contracts can be called by any client in the network, including malicious ones that try to force the contract to deviate from the intended behavior [36]. In recent years, there have been several large attacks exploiting flawed program flow control in smart contracts. Most famously, attackers managed to steal around 50 M USD [23, 36] from a decentralized autonomous organization, the DAO. In two attacks on the Parity multi-signature wallet, attackers stole cryptocurrencies worth 30 M USD [12] and froze 150 M USD [65].

Programming dApps. In this paper, we explore a programming model that ensures the correctness and security of the client–contract interaction of dApps by-design. Deviations from the intended interaction protocols due to implementation errors and/or malicious attacks are a critical threat (besides other issues such as arithmetic or buffer overflows, etc.) as demonstrated e.g., by the DAO attack [23, 36] mentioned above.

dApps are multi-party applications. For such applications, there are two options for the programming model: a *local* and a *global model*. In a *local model*, parties are defined each in a separate *local* program and their interactions are encoded via effectful send and receive instructions. Approaches that follow this model stem from process calculi [46] and include actor systems [2] and approaches using session types [27], and linear logic [86]. In contrast, in a *global model*, there is a single program shared by both parties and interactions are encoded via combined send-and-receive operations with no effects visible to the outside world. This model is represented by tierless [16, 22, 38, 69, 70, 80, 81, 87] and choreographic [40, 47, 59] languages. The local model requires an explicitly specified protocol to ensure that every send effect has a corresponding receive operation in an interacting – separately defined – process. With a global model, there is no need to separately specify such a protocol. All parties run the same program in lock-step, where a single send-and-receive operation performs a send when executed by one party and a receive by the other party. Due to encapsulating communication effects, there is no non-local information to track – the program's control flow defines the correct interaction and a simple type system is sufficient.

Current approaches to dApp programming – industrial or research ones – follow a local model, Contract and client are implemented in separate programs, thus safety relies 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 uses Solidity [58] for the contract and JavaScript for clients. Solidity relies on developers following best practices recommending to express the protocol as runtime assertions integrated into the contract code [33]. Failing to correctly introduce assertions may give parties illegal access to monetary values to the detriment of others [52, 60].

The currently dominant encoding style of the protocol as *finite state machine (FSM)* uses one contract-side function per FSM transition [18–20, 58, 75, 76]. While FSMs model a useful class of programs that can be efficiently verified, writing programs in such style directly has several shortcomings. First, an FSM corresponds to a control-flow graph of basic blocks, which is low-level and more suited as an internal compiler representation than as a front-end language for humans. Second, with the FSM style, the contract is a passive entity whose execution is driven by clients. This design puts the burden of enforcing the protocol on the programmers of the contract, as they have to explicitly consider in what state which messages are valid and reject all invalid messages from the clients. Otherwise, malicious clients would be able to force the contract to deviate from

[1] 700 K to 2.7 M contracts have been deployed per month between July 2020 and June 2021 [43] on the Ethereum blockchain – the most popular dApps platform [24]. Some dApps manage tremendous amounts of assets, e.g., Uniswap [85] – the largest Ethereum trading platform had a daily trading volume of 0.5 B – 1.5 B USD in June 2021.

its intended behavior by sending messages that are invalid in the current state. Third, ensuring protocol compliance statically to guarantee safety requires advanced types, as the type of the next action depends on the current state.

In research, some smart contract languages [9, 18–20, 25, 61, 75, 76] have been proposed to overcome the FSM-style shortcomings. They rely on advanced type systems such as session types, type states, and linear types. There, processes are typed by the protocol (of side-effects such as sending and receiving) that they follow and non-compliant processes are rejected by the type-checker.

The global model has not been explored for dApp programming – which 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 as well.

Prisma. Prisma enables interleaving contract and client logic within the same program and adopts a *direct style (DS)* notation for encoding send-and-receive operations akin to languages with baked-in support for asynchronous interactions, e.g., via async/await [8, 73]. Prisma leaves it to the compiler to map down high-level declarative DS to low-level FSM style. It avoids the need for advanced typing discipline and allows the contract to actively ask clients for input, promoting an execution model where a dominant acting role controls the execution and diverts control to other parties when their input is needed, which matches well the dApp setting.

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

For this, 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 to suspend execution and resume later – which could be used, otherwise, to implement asynchronous communication. Given that CPS translations reify control flow, without proper guarding, malicious clients could force the contract to deviate from the intended flow by passing a spoofed value to the contract. Thus, it is imperative to prove that our *distributed CPS translation* ensures control-flow integrity of the contract, which we do on top of a formal definition of the compilation steps. The formally proven secure Prisma compiler eliminates the risk of programmers implementing unsafe interactions that can potentially be exploited.

Contributions. We make the following contributions:

(1) We introduce Prisma,[2] a global language for tierless dApps with direct-style client–contract interactions and explicit access control, implemented as an embedded DSL in Scala. Crucially, Prisma automatically enforces the correct program flow (Section 2).
(2) A core calculus, MiniPrisma, which formalizes both Prisma and its compiler, as well as a proof that our compiler guarantees the preservation of control flow in presence of an attacker that controls the client code (Section 3).
(3) Case studies which show that Prisma can be used to implement common applications without prohibitive performance overhead (Section 5).
(4) A comparison of Prisma with a session type and a type state smart contract programming language and the mainstream Solidity/JavaScript programming model (Section 6).

[2] Prisma implementation and case studies are publicly available: https://github.com/stg-tud/prisma

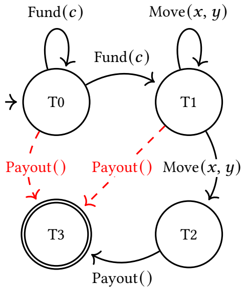


Fig. 1. TicTacToe control flow.

Table 2. Location annotations.

Annotations	Description
@co	on contract
@cl	on clients
@co @cl	independent copies on clients and contract
@co @cross	on contract, but also accessible by client
@cl @cross	(illegal combination)

```
@prisma object TicTacToeModule {

  @co @cl case class UU(x: U8, y: U8)

  class TicTacToe(
    val players: Arr[Address],
    val fundingGoal: Uint) {

    // u8 is an unsigned 8-bit integer
    @co @cross var moves: U8 = "0".u8
    @co @cross var winner: U8 = "0".u8
    @co @cross val board: Arr[Arr[U8]] =
      Arr.ofDim("3".u, "3".u)

    @co def performMove(x: U8, y: U8): Unit =
      { /* ... */ }
    @cl def updateBoard(): Unit =
      { /* ... */ }
    @cl def fund(): (U256, Unit) =
      (readLine("How much?").u, ())
    @cl def move(): (U256, UU) =
      ("0".u, UU(readLine("x-pos?"),
                 readLine("y-pos?"))
    @cl def payout(): (U256, Unit) = {
      readLine("Press (enter) for payout")
      ("0".u, ())
    }
    @co val init: Unit = {
      while (balance() < FUNDING_GOAL) {
        awaitCl(_ => true) { fund() }
      }
      while (moves < "9".u && winner == "0".u) {
        val pair: UU = awaitCl(a =>
          a == players(moves % "2".u)) { move() }
        performMove(pair.x, pair.y)
      }
      awaitCl(a => true) { payout() }
      if (winner != "0".u) {
        players(winner - "1".u).transfer(balance())
      } else {
        players("0".u).transfer(balance() / "2".u)
        players("1".u).transfer(balance()) // remainder
      }
    }
  }
}
```

Fig. 3. TicTacToe dApp.

2 PRISMA IN A NUTSHELL

We present Prisma by the example of a TicTacToe game, demonstrating that client and contract are written in a single language, where protocols are expressed by control flow (instead of relying on advanced typing disciplines) and enforced by the compiler.

Example. TicTacToe is a two-player game over a 3 × 3 board. Players take turns in writing their sign into one of the free fields until all fields are occupied, or one player wins by owning three fields in any row, column, or diagonal. The main transaction of a TicTacToe dApp is `Move(x,y)` used by a player to occupy field `(x,y)`. A `Move(x,y)` is valid if it is the sender's turn and `(x,y)` is empty. Before the game, players deposit their stakes, and after the game, the stakes are paid to the winner.

Fig. 1 depicts possible control flows with transitions labeled by client actions that trigger them. Black arrows depict intended control flows. The dApp starts in the funding state where both parties deposit stakes via *Fund*(c). Next, parties execute *Move*(x, y) until one party wins or the game ends in a draw. Finally, any party can invoke a payout of the stakes via *Payout*().[3] Red dashed arrows illustrate the effects of a mismanaged control flow: a malicious player could trigger a premature payout preventing the counterpart to get financial gains.

Tierless dApps. Prisma is implemented as a DSL embedded into Scala, and Prisma programs are also valid Scala programs.[4] Prisma interleaves contract and client logic within the same program. Annotations `@co` and `@cl` explicitly place declarations on the contract and on the client, respectively (cf. Tab. 2). A declaration marked as both `@co` and `@cl` has two copies. For security, code placed in one location cannot access definitions from the other — an attempt to do so yields a compile-time error. Developers can overrule this constraint to enable clients to read contract variables or call contract functions by combining `@co` with `@cross`. Combining `@cl` with `@cross` is not allowed – information can only flow from client to contract as part of a client–contract interaction protocol.

There are three kinds of classes. *Located classes* are placed in one location (annotated with either `@co` or `@cl`); they cannot contain located members (annotated with either `@co` or `@cl`) and their instances cannot cross the client–contract boundary, e.g., be passed to or returned from `@cross` functions. *Portable classes* are annotated with both `@co` and `@cl`. Their instances can be passed to and returned from `@cross` functions; they must not contain mutable fields. *Split classes* have no location annotation; their instances live partly in both locations; they cannot be passed to or returned from `@cross` functions and their members must be located.

Prisma code is grouped into modules. While client declarations can use and be used from standard (non-Prisma) Scala code, contract declarations are not accessible from Scala, and can only reference contract code from other Prisma modules (because contract/client code lives in different VMs).

For illustration, consider the TicTacToe dApp (Fig. 3). The `TicTacToeModule` (Line 1) – modules are called `object` in Scala – contains a portable class `UU` (Line 3) and a split class `TicTacToe` (Line 5). Variables `moves`, `winner`, `board` (Lines 10, 11, 13) are placed on the contract and can be read by clients (`@co @cross`). The `updateBoard` function (Line 17) is placed on the client and updates client state (e.g., client's UI). The `move` function (Line 15) is placed on the contract and changes the game state (`move`). `move` is not annotated with `@cross`, because `@cross` is intended for functions that do not change contract state and can be executed out-of-order without tampering with the client–contract interaction protocol. While Scala only has signed integers and signed longs literals, these are uncommon in Ethereum. Therefore, Prisma provides portable unsigned and signed integers for power-of-two bitsizes between 2^3 to 2^8, with common arithmetic operations, e.g., `"0".u8` is an unsigned 8-bit integer of value 0 (Line 10).

Encoding client–contract protocols. In Prisma, a client-contract protocol is encoded as a split class containing dedicated `awaitCl` expressions for actively requesting and awaiting messages from specific clients and standard control-flow constructs. Hence, creating a new contract instance corresponds to creating a new instance of a protocol; once created, the contract instance actively triggers interactions with clients. The `awaitCl` expressions have the following syntax:

```
def awaitCl[T](who: Addr => Bool)(body: => (Ether, T)): T
```

They take two arguments. The first (`who`) is a predicate used by clients to decide whether it is their turn and by the contract to decide whether to accept a message from a client. This is unlike Solidity, where a function may be called by any party by default. By forcing developers to explicitly define access control, Prisma reduces the risk of human failure. The second argument (`body`) is the expression to be executed by the client. The client returns a pair of values to the contract: the

[3] We omit handling timeouts on funding and execution for brevity.

[4] In Scala `val`/`var` definitions are used for mutable/immutable fields and variables, `def` for methods, `class` for classes, and `object` for singletons. A `case class` is a class whose instances are compared by structure and not by reference.

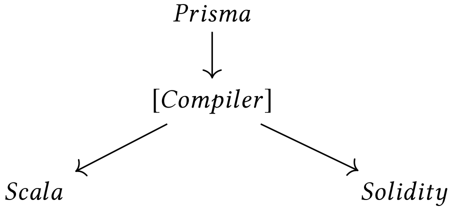
amount of Ether and the message. The former can be accessed by the contract via the built-in expression **`value`**, the latter is returned by **`awaitCl`**. Besides receiving funds via **`awaitCl`**, a contract can also check its current balance (**`balance`**`()`), and transfer funds to an account (`account.`**`transfer`**`(amount)`)).

Prisma's programming model is specifically designed to accommodate blockchain use cases. In contrast to other tierless models like client-server, we emphasise inversion of control such that the code is written as if the contract was the active driver of the protocol, while clients are passive and only react to requests by the contract. This enables to enforce the protocol on the contract side. For this reason, for example, we support the **`awaitCl`** construct on the active contract side whereas there is no corresponding construct on the passive client side.

For illustration, consider the definition of `init` on the right-hand side of Fig. 3, Line 28. It defines the protocol of `TicTacToe` as follows. From the beginning of `init`, the flow reaches **`awaitCl`** in Line 30 where the contract waits for clients to provide funding (by calling `fund`). Next, the contract continues until **`awaitCl`** in Line 33 and clients execute `move` (Line 21) until the game ends with a winner (`winner != 0`) or a draw (`moves >= 9`). At this point – **`awaitCl`** in Line 37 – any party can request a payout and the contract executes to the end. The example illustrates how direct-style **`awaitCl`** expressions and the tierless model enable encoding multiparty protocols as standard control flow, with protocol phases corresponding to basic blocks between **`awaitCl`** expressions.

Compiling Prisma to Solidity. Abstractly, Prisma's compiler takes a Prisma dApp program and splits it into two separate programs: A Scala client program and a Solidity contract program (Fig. 4a). In more detail, the compiler (1) places all definitions according to their annotations and (2) splits contract methods that contain **`awaitCl`** expressions into a method that contains the code up to the **`awaitCl`** and a method that contains the continuation after the **`awaitCl`** (taking the result of the **`awaitCl`** as an argument). Once deployed, a contract is public and can be messaged by arbitrary clients – not exclusively the ones generated by Prisma – hence, we cannot assume that clients will actually execute the body passed to them by an **`awaitCl`** expression. To cope with malicious clients trying to tamper with the control flow of the contract, the compiler hardens contract code by generating code to enforce *control flow integrity*: storing the current phase before giving control to the client and rejecting methods invoked by wrong clients or in the wrong phase.

For illustration, the code generated from Fig. 3 is schematically shown in Fig. 4b and 4c. The methods `updateBoard`, `fund`, `move`, and `payout` are annotated **`@cl`** and thus compiled into the client program (Fig. 4b). The variables `moves`, `winner` and `board`, and the method `performMove` are annotated **`@co`** and thus compiled into the contract program (Fig. 4c). Further, three new methods are generated on both the client and the contract – one for each **`awaitCl`** expression in `init` – corresponding to phases in the logical protocol (Fig. 1). The `Funding` method of the client (Line 16) is generated from the body of the first **`awaitCl`**. Similarly, the `Move` method (Line 18) is generated from the second **`awaitCl`** and the `Payout` method (Line 20) from the third **`awaitCl`**. In the example, the generated methods are given meaningful names by capitalizing the single method called in the body of the **`awaitCl`** expressions form which they were generated. In the actual implementation, generated methods are simply enumerated. The code up to the first **`awaitCl`** (Line 30, Fig. 3) is placed in the constructor of the generated contract, which ends by setting the active phase to `Funding`. The code between the first and the second **`awaitCl`** either loops back to the first **`awaitCl`** or continues to the second one (Line 33). The code is placed in the `Fund` method that requires the phase to be `Funding`, and may change it to `Exec` if the loop condition fails. Similarly, the method `Move` is generated to contain the loop between the second and the third **`awaitCl`** (Line 37); and the method `Payout` contains the code from the third **`awaitCl`** to the end of `init`. Only the second **`awaitCl`** contains a (non-trivial) access control predicate, which results in an additional assertion in the body of `Move` (Line 46, Fig. 4a). Observe that the



(a) Compilation scheme

```
class TTT {

  // @cl annotated definitions
  def updateBoard(): Unit =
    { /* ... */ }
  def fund(): (U256, Unit) =
    (readLine("How much?").u, ())
  def move(): (U256, UU) =
    ("0".u, UU(readLine("x-pos?"),
    readLine("y-pos?"))
  def payout(): (Ether, Unit) = {
    readLine("Press (enter) for payout")
    ("0".u, ()) }

  // body of awaitCl expressions
  def Fund(): (Ether, Unit) =
    fund()
  def Move(): (Ether, UU)   =
    move()
  def Payout(): (Ether, Unit) =
    payout()

  /* ... */




}
```

(b) Scala client

```
contract TTT {
  State phase = T0; enum State {T0, T1, T2, T3}

  // @co annotated definitions
  int moves  = 0;
  int winner = 0;
  int[][] board;
  function peformMove(int x, int y) private { /*...*/ }

  // continuation of awaitCl expressions
  function Fund() public {
    require(phase == T0);
    /*...*/;
    if (!(balance < FUNDING_GOAL)) phase = T1;
    /* else phase remains T0; this models the first while loop */
  }
  function Move(int x, int y) public {
    require(phase == T1 && sender == players(moves % 2));
    /*...*/;
    if (!(moves < 9 && winner == 0)) phase = T2;
    /* else phase remains T2; this models the second while loop */
  }
  function Payout() public {
    require(phase == T2);
    /*...*/;
    phase = T3;
  }
}
```

(c) Solidity contract

Fig. 4. TicTacToe dApp after compilation, simplified

return types of the generated client methods are the argument types of the corresponding contract methods.

Compilation Techniques. While CPS is a key step in our translation pipeline, the example shows the final defunctionalised, trampolined code. The final output does not contain explicit continuations (i.e., a function that takes another function as an argument and calls that as its continuation). Instead, after defunctionalizing and trampolinizing the CPS translation, only one top-level function (`Fund`, `Move`, `Payout`) is callable at each `phase`, which is ensured by the `require` statement at the beginning of each function, and each function sets the next `phase` at the end. These functions play the role of the continuations.

Let us look at the correspondence between the original Prisma code (Fig. 3) and the generated Solidity code (Fig. 4c) from a higher-level perspective: To verify that the Prisma code matches the generated Solidity code, we proceed as follows.

First, we verify that the control flow of Fig. 3 is accurately described by the automaton diagram in Fig. 1. In particular, we observe that there are two loops in the automaton and there are also two `while` loops in the Prisma code. Further, there are three `awaitCl` expressions in the code, and there are three states in the automaton (plus a final state).

Second, we verify that the automaton in Fig. 3 corresponds to the program flow of the Solidity code in Fig. 4c. In particular, we observe that there are four states in the automaton and there are four states in the Solidity code. Three of those have an associated function (`T0` is `Fund`, `T1` is `Move`, `T2` is `Payout`), which are the only public functions that can be invoked in that state, thanks to the `require` statements. In the final state `T3`, no public function can be invoked. Furthermore, we can see that the automaton has two loops. It is possible to go from `T0` either to `T1` or stay in `T0`. This is represented in the Solidity code, by checking for the loop condition at the end of the function associated to `T0`, and then either changing the phase to `T1`, or doing nothing, which means staying in `T0`. Similarly, the loop in state `T1` is encoded with an `if` at the end of the function to conditionally move to the next phase.

These two steps should illustrate how the control flow of the Prisma program – which is abstractly visualized by the automaton – is implemented and enforced by the generated Solidity program.

3 COMPILATION AND ITS CORRECTNESS

We informally introduce Prisma's compilation process and our notion of correctness before formally specifying and proving the compiler correct.

3.1 High-level Overview of Prisma's Secure Compilation

To implement the contract-client interaction, we CPS-translate Prisma code and execute continuations alternately between contract and client. A standard CPS translation is, however, not sufficient because the control flow is distributed and we need to send function calls (i.e., the current continuation) over the network – or, more specifically, send the name and the arguments of the next function to execute. For this, we defunctionalise [71] the code to turn functions calls (which represent continuations) into data. This compilation process performs an *inversion of control* between the contract and the client. With Prisma's contract–client communication in direct style, we can write dApps as if *the contract* was in control of the execution; Prisma allows the contract to request messages from clients and to process only responses that it requested.

After the compilation process, clients are in control of the execution because, in blockchains, contracts purely respond to messages from clients. As a result, dApps may become the target of malicious attacks. In our security model, we trust the contract to execute the code that we generate for it, whereas we consider the client code untrusted, i.e., the client side can run arbitrary code. Crucially, it could pass unintended continuations to the contract to force the execution to continue in an arbitrary state. For example, in the source code of the TicTacToe game (Section 2), one needs to go through the game loop after funding and before payout. Yet, the compiled code is separated and distributed into small chunks. Parties execute a chunk and then wait for other parties to decide on a move that influences how to proceed with the execution. For this reason, the client could send a message at any time telling the contract to go into the payout phase. We need to guard the contract against such attempts to make it deviate from the protocol. Conceptually, if the client was able to force the execution to continue in an arbitrary state, the control flow in the Prisma source would be violated. Execution would 'jump' from one client expression to another one skipping the code in between, which is not possible with the semantics of the source language.

$$\begin{tikzcd}
\text{source} & \text{target} \\
\mathsf{Term_s} \arrow[r, "compile"'] \arrow[d, dotted, "eval_s"] \arrow[dd, dotted, bend right=50, "eval_{s,b}"'] & \mathsf{Term_t} \arrow[d, dashed, "eval_t"'] \arrow[dd, bend left=50, "eval_{t,b}"] \\
\mathsf{Trace_s} \arrow[r, leftrightarrow, "correctness"'] & \mathsf{Trace_t} \\
\mathsf{Trace^*_s} \arrow[r, leftrightarrow, "security"'] & \mathsf{Trace^*_t}
\end{tikzcd}$$

Fig. 5. Secure Compilation

Prisma's compiler avoids such attacks and preserves control flow by inserting guards on the contract side. Guards are in places where the basic blocks of the program have been separated and distributed onto different hosts by CPS translation – to reject any improper continuations from clients. Guarding ensures the control flow integrity [1] of the contract in the presence of malicious clients by excluding any behavior of the compilation target that cannot be observed from the source. Informally, this is our notion of *secure compilation*, which we rigorously define and prove for Prisma's compiler in this section. The compilation process is key in hiding the complexity of enforcing distributed control flow from the developer – hence, a formal proof of its correctness is critical.

To formalize the compiler, we specify a source and a target language. Fig. 5 shows a schema of our compilation and the proof. The compiler (Fig. 5, top) is a function that maps terms in the source language ($Term_s$) into terms in the target language ($Term_t$). A correct compiler preserves some properties of the code – depending on the notion of correctness. For example, typeability-preserving and semantics-preserving compilers have been extensively studied [63]. Because types are not the focus of this paper, we omitted them from the figure. In the middle part of Fig. 5, we show the evaluation of source and target to traces ($eval_s$ and $eval_t$, respectively) – and traditional compiler correctness as the equivalence between traces generated from the sources ($Trace_s$) and from the target ($Trace_t$). But compiler correctness[5] in this traditional sense is not sufficient in the presence of malicious attackers that can tamper with parts of the code. Instead, we need to prove that Prisma is a *secure abstraction*, i.e., if security problems can arise on the target, they must be visible in the Prisma source code, too, so that developers do not need to look at target code to reason about potentially misbehaving clients. To this end, we define a hypothetical *attacker model on the source code* as the ability to only replace the body of a Prisma client expression and show that, with the contract part hardened with guards, the target attacker does not gain additional power over the hypothetical source attacker. Specifically, we define malicious semantics $eval_{t,b}$ and $eval_{s,b}$ for the target and the source language, respectively, and show that $eval_{t,b}(compile\ e) = compile(eval_{s,b}\ e)$ (security property in Fig. 5).

In the reminder of this section:

- We present the core calculus (Section 3.2) $\mathsf{MiniPrisma}_*$ – a hybrid language that includes elements of both the source ($\mathsf{MiniPrisma}_s$) and the compilation target ($\mathsf{MiniPrisma}_t$), while abstracting over details of both Scala and Solidity. We define a hybrid language because the source and the target share many constructs – the hybrid language allows us to focus on how the differences are compiled.

[5] Type and semantics preservation is not the focus of this paper; we presume them for our compiler without a formal proof.

$$id \in ID \qquad i \in I \qquad j \in \{\text{who}, \text{state}, \text{clfn}, \text{cofn}\}$$

$$\begin{array}{llll}
\text{(definition)} & d & ::= & @\text{co this}.i = v;\ d \mid @\text{cl this}.i = v;\ d \mid () \\
\text{(synthetic definition)} & b & ::= & @\text{co this}.j = v;\ b \mid @\text{cl this}.j = v;\ b \mid () \\
\text{(program)} & P & ::= & d;\ b;\ m \\
 & & & \\
\text{(constant)} & c & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \text{true} \mid \text{false} \mid () \mid \&\& \mid + \mid == \mid < \mid \text{try} \\
 & & & \mid \text{»=} \mid \text{trmp} \mid \text{Done} \mid \text{More} \\
\text{(value)} & v & ::= & c \mid v :: v \mid x \longrightarrow e \\
\text{(pattern)} & x & ::= & c \mid x :: x \mid id \\
\text{(expression)} & e & ::= & c \mid e :: e \mid x \longrightarrow e \mid id \mid x{=}e;\ e \mid e\ e \\
 & & & \mid \text{this}.i \mid \text{this}.i := e \mid \text{this}.j \mid \text{this}.j := e \\
\text{(main expression)} & m & ::= & c \mid m :: m \mid x \longrightarrow e \mid id \mid x{=}m;\ m \mid m\ m \\
 & & & \mid \text{this}.i \mid \text{this}.i := m \mid \text{this}.j \mid \text{this}.j := m \\
 & & & \mid \text{awaitCl}_s(e, () \longrightarrow e) \mid \text{awaitCl}_t(c, () \longrightarrow e)
\end{array}$$

Fig. 6. MiniPrisma$_*$ syntax.

- We model the compiler (Section 3.3) as a sequence of steps that transform MiniPrisma$_*$ programs via several intermediate representations.
- We define MiniPrisma$_*$ semantics as a reduction relation over configurations consisting of traces of evaluation events and expressions being evaluated (Section 3.4). We distinguish between a good semantics, which evaluates the program in the usual way, and a bad semantics, which models attackers by ignoring client instructions and producing arbitrary values that are sent to the contract.
- We prove secure compilation by showing that the observable behavior of the programs before and after compilation is equivalent (Section 3.5). We capture the observable program behavior by the trace of events generated during program evaluation (as guided by the semantic definition) and show trace equivalence of programs before and after compilation.

3.2 Syntax

The syntax of MiniPrisma$_*$ (Fig. 6), has three kinds of identifiers id, i, j, from unspecified sets of distinct names. Pure identifiers id are for function arguments and let bindings; mutable variables i are for heap variable assignment and access. In the target program, mutable variables j (who, state, clfn, cofn) generated by the compiler can also appear. We call compiler-generated identifiers *synthetic*. Normal identifiers are separated from synthetic ones to distinguish compiler generated and developer code. Definitions d and definitions for synthetic identifiers b are semicolon-separated lists of declarations that assign values to variables and annotate either the contract or the client location. Each program P consists of definitions d and synthetic definitions b followed by the main contract expression m. Program P corresponds to a single Prisma split class, d and b to methods and generated methods, and m to a constructor containing the initialisation of its class members (such as the body of `init`, Fig. 3).

Constants c are unsigned 256 bit integer literals and built-in operators. MiniPrisma$_*$ supports tuples introduced by nesting pairs (::) and eliminated by pattern matching. Tuples allow multiple values to cross tiers in a single message. Values v are constants, value pairs, and lambdas. Patterns x are constants, pattern pairs, and variables. Expressions e are constants, expression pairs, lambdas, variables, variable accesses/assignments, bindings and function applications.

$$\begin{array}{rcl}
m_0\ c\ m_1 & = & c(m_0, m_1) \\
(m_0, ..., m_n) & = & m_0 :: ... :: m_n :: () \\
m_0;\ m_1 & = & () = m_0;\ m_1 \\
\mathsf{assert}(m_0);\ m_1 & = & \mathsf{true} = m_0;\ m_1 \\
x \leftarrow e_1;\ m_2 & = & x = \mathsf{awaitCl}_t(() \rightarrow e_1);\ m_2 \\
\mathsf{if\ let}\ x = m_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & = & \mathsf{try}(m_1, x \rightarrow e_2, () \rightarrow e_3)
\end{array}$$

Fig. 7. Syntactic sugar.

Main expressions m may further contain remote client expressions, embedding client code into contract code and waiting for its result. The source client expression $\mathsf{awaitCl}_s(e,\ () \rightarrow e)$ can be answered by any client whose address fulfills the predicate specified as first argument. $\mathsf{awaitCl}_s$ corresponds to direct-style remote access via **`awaitCl`** in Prisma. We use the syntax form $\mathsf{awaitCl}_t(c,\ () \rightarrow e)$ to model the execution of code e on the specified client c. $\mathsf{awaitCl}_t$ has no correspondence in the source syntax. Our compilation first splits the predicate from the source client expressions into a separate access control guard. Then, it eliminates client expressions, turning the contract into a passive entity that stops and waits for client input.

We now map the hybrid language $\mathsf{MiniPrisma}_*$ to the source and target languages, $\mathsf{MiniPrisma}_s$ and $\mathsf{MiniPrisma}_t$. $\mathsf{MiniPrisma}_s$ has all expressions of $\mathsf{MiniPrisma}_*$, except those that contain $\gg\!=$ (bind), trmp (trampoline), Done, More, $\mathsf{awaitCl}_t$, or synthetic identifiers j. $\mathsf{MiniPrisma}_t$ has all expressions of $\mathsf{MiniPrisma}_*$ except those that contain $\mathsf{awaitCl}_s$, $\mathsf{awaitCl}_t$, $\gg\!=$.

$\gg\!=$ and $\mathsf{awaitCl}_t$ may not appear neither in source nor target programs; the former is used only as an intermediate construction for the compiler, the latter only during evaluation to track the current location.

Syntactic sugar. In Fig. 7, we define some syntactic sugar to improve readability. We use infix binary operators and tuple syntax for nested pairs ending in the unit value (); we elide the let expression head for let bindings matching (), $\mathsf{assert}(x)$ is a let binding matching true; we use monadic syntax for let bindings of effectful expressions; if let $x = m$ then e else e is the application of the built-in try function.

Events and configurations. In Fig. 8, we define left-to-right evaluation contexts E [34]; and compilation frames F [66], such that every expression decomposes into a frame-redex pair $F\ e$ or is an atom a. Events p and q are lists that capture the observable side-effects of evaluating expressions. They are either (a) state changes $\mathsf{wr}(c, i, v)$ and $\mathsf{wr}(c, j, v)$, from the initial definitions or variable assignment, where i and j are the variable being assigned, c the location, and v the assigned value, or (b) client-to-contract communication $\mathsf{msg}(c, v)$, where c is the address of the client and v the sent value. Configurations $C = p; q; cm$, represent a particular execution state, where p (and q) are traces of normal (and synthetic) events produced by the evaluation, c is the evaluating location, and m is the expression under evaluation.

Initialization. Initialization in Fig. 9 generates the initial program configuration, which models the decentralized application with a single contract and multiple clients. We model a fixed set of clients A interacting with a contract. The initialization of a program $d; b; m$ to a configuration $p; q; 0; m$ leaves the expression m untouched and generates a list of events – one write event for each normal and synthetic definition. Location 0 represents the contract.

$$
\begin{array}{lcll}
\text{(frame)} & F & ::= & \mathsf{awaitCl}_s(\square, () \to e) \mid \square\ e \mid e\ \square \mid \square :: e \mid e :: \square \\
 & & \mid & x = \square;\ e \mid x = e;\ \square \mid \mathsf{this}.i := \square \mid \mathsf{this}.j := \square \\
\text{(atom)} & a & ::= & \mathsf{this}.i \mid \mathsf{this}.j \mid c \mid id \mid x \to e \\
\text{(context)} & E & ::= & \square \mid E :: m \mid v :: E \mid E\ m \mid v\ E \mid x = E; m \mid \mathsf{this}.i := E \mid \mathsf{this}.j := E \\
\text{(event)} & p & ::= & \mathsf{wr}(c, i, v)\ p \mid \mathsf{msg}(c, v)\ p \mid () \\
\text{(synthetic event)} & q & ::= & \mathsf{wr}(c, j, v)\ q \mid () \\
\text{(configuration)} & C & ::= & p;\ q;\ c;\ m
\end{array}
$$

Fig. 8. Frames, Events and configurations.

$$
\begin{array}{lcl}
init_A(d;b;m) & = & init_A(d;b);\ 0;\ m \\
init_A(d;b) & = & (wr(0,i,v) \mid \forall\ (@\mathsf{co}\ \mathsf{this}.i = v) \in d) \\
 & & (wr(0,j,v) \mid \forall\ (@\mathsf{co}\ \mathsf{this}.j = v) \in b) \\
 & & (wr(c,i,v) \mid \forall\ (@\mathsf{cl}\ \mathsf{this}.i = v) \in d,\ c \in A) \\
 & & (wr(c,j,v) \mid \forall\ (@\mathsf{cl}\ \mathsf{this}.j = v) \in b,\ c \in A)
\end{array}
$$

Fig. 9. Initialization.

3.3 Compilation

The compiler eliminates language features not supported by the compilation target one by one, lowering the abstraction level from (1) *direct style communication (DS)* – which needs language support for !-notation [10] – through the intermediate representations of (2) *monadic normal form (MNF)* – which needs support for do-notation [53] – and (3) *continuation-passing style (CPS)* – which needs higher-order functions – to (4) explicitly encoding *finite state machines (FSM)* – for which first-order functions suffice. In the following, we provide an intuition for the compiler steps and subsequently their formal definitions.

First, the compilation steps *mnf* and *assoc* transform DS remote communication $\mathsf{awaitCl}_s(e, () \to e)$ to variable bindings ($id := e$) and nested let bindings are flattened such that a program is prefixed by a sequence of let expressions. Second, step *guard* generates access control guards around client expressions to enforce correct execution even when clients behave maliciously. Third, step *cps* transforms previously generated let bindings for remote communication ($x \leftarrow e_1;\ m_2$) to monadic bindings $e_1 \gg\!= x \to m_2$. Fourth, step *defun* transforms functions into data structures that can be sent over the network and are interpreted by a function (i.e., an FSM) on the other side. Compared to standard defunctionalization, we handle two more issues. First, we defunctionalize the built-in higher-order operator ($\gg\!=$) by wrapping the program expression into a call to a trampoline $\mathsf{trmp}(...)$ and transforming the bind operator ($... \gg\!= x \to ...$) to the $(\mathsf{More}, ..., ...)$ data structure; the trampoline repeatedly interprets the argument of More until it returns Done instead of More signaling the program's result. Second, we keep contract and client functions separate by generating separate synthesized interpreter functions, called cofn and clfn, thereby splitting the code into the parts specific to contract and client.

MNF transformation (Fig. 10). The *mnf′* function wraps the main expression m into a call to the trampoline with the pair (Done, m) – signaling the final result – as argument. Then, *mnf* transforms expressions recursively, binding sub-expressions to variables, resulting in a program prefixed by a sequence of let bindings. As recursive calls to *mnf* may return chains of let bindings, we apply *assoc* to produce a flat chain of let bindings. Given a let binding, whose sub-expressions are in MNF, associativity recursively flattens the expression, by moving nested let bindings to the

$$
\begin{array}{ll}
\mathit{mnf}'(d; b; m) & = d; b; \mathrm{trmp}(\mathit{mnf}((\mathrm{Done}, m))) \\
\mathit{mnf}(F\ e) & = \mathit{assoc}(\mathit{id}_0{=}e;\ \mathit{mnf}(F\ \mathit{id}_0)) \\
\mathit{mnf}(a) & = a \\[1ex]
\mathit{assoc}(x_0{=}(x_1{=}m_1;\ m_0);\ m_2) & = \mathit{assoc}(x_1{=}m_1;\ \mathit{assoc}(x_0{=}m_0;\ m_2)) \\
\mathit{assoc}(m) & = m
\end{array}
$$

Fig. 10. Monadic normal form transformation.

$$
\begin{array}{lcl}
\mathit{guard}'(d; b; \mathrm{trmp}(m)) & = & d; b; \mathrm{trmp}(\mathit{guard}(m)) \\[2ex]
\mathit{guard}\left(\begin{array}{l} x \leftarrow_s (e_0, () \rightarrow e_1); \\ m_2 \end{array}\right) & = & \left(\begin{array}{l}
\mathrm{this.\,who} := e_0;\ \mathrm{this.\,state} := c; \\
x \leftarrow_s (() \rightarrow \mathrm{true}, () \rightarrow e_1); \\
\mathrm{assert}(\mathrm{this.\,state} == c\ \&\& \\
\qquad\quad \mathrm{this.\,who}(\mathrm{this.\,sender})); \\
\mathrm{this.\,state} := 0;\ \mathit{guard}(m_2)
\end{array}\right) \\
 & & \ \mathrm{where}\ c\ \ \mathrm{fresh} \\
\mathit{guard}(x = e_0;\ m_1) & = & x = e_0;\ \mathit{guard}(m_1) \\
\mathit{guard}(m) & = & m
\end{array}
$$

Fig. 11. Guarding.

$$
\begin{array}{lcl}
\mathit{cps}'(d; b; \mathrm{trmp}(m)) & = & d; b; \mathrm{trmp}(\mathit{cps}(m)) \\[1ex]
\mathit{cps}(x \leftarrow_s (() \rightarrow \mathrm{true}, e_0);\ m_1) & = & e_0 \gg\!= (x \rightarrow \mathit{cps}(m_1)) \\
\mathit{cps}(x = e_0;\ m_1) & = & x = e_0;\ \mathit{cps}(m_1) \\
\mathit{cps}(m) & = & m
\end{array}
$$

Fig. 12. Continuation-passing style transformation.

front, $(\ldots\ (\ldots\ m_0;\ m_1);\ m_2 = \ldots\ m_0;\ (\ldots\ m_0;\ m_2))$, creating a single MNF expression (i.e., *assoc* is composition for MNF terms).

Guarding (Fig. 11). We insert access control guards for remote communication expressions $\leftarrow_s$ to enforce (i) the execution order of contract code after running the client expression and (ii) that the correct client invokes the contract continuation. The transformation sets the synthetic variable state to a unique value before the client expression, and stores the predicate to designate valid clients in the synthetic variable who. After the client expression, the generated code asserts that the contract is in the same state, and checks that the sender fulfills the predicate. The assertion trivially holds in the sequential execution of the source language, but after more compilation steps the client will be responsible for calling the correct continuation on the contract. Since client code is untrusted, the contract needs to ensure that only the correct client can invoke only the correct continuation.

CPS transformation (Fig. 12). The *cps* transformation turns the chains of let bindings produced by *mnf* into CPS. The chain contains three cases of syntax forms: (1) monadic binding $(x \leftarrow \ldots;\ m_1)$, (2) let binding $(x = e_0;\ m_1)$, or (3) final expression. For (1), *cps* replaces the monadic binding with an explicit call to the bind operator $(\ldots \gg\!= (x \rightarrow \mathit{cps}(m_1)))$. For (2) and (3), *cps* recurses into the tail of the chain. This resembles do-notation desugaring (e.g., in Haskell).

$$\begin{array}{lcl}
\mathit{defun}'(d; b; e) & = & \mathit{defun}(d; \mathit{coclfn}(b, \mathit{id}, \mathsf{assert}(\mathsf{false}), \mathsf{assert}(\mathsf{false})); e) \\
 & & \quad \text{where} \quad \mathit{id}\ \text{fresh} \\[1em]
\mathit{defun}\left(\begin{array}{l} d; \mathit{coclfn}(b, \mathit{id}, \\ \quad e_{1,alt}, \\ \\ \quad e_{2,alt} \\ \quad); \\ (() \rightarrow e_1) \gg\!= (x \rightarrow e_2) \end{array}\right) & = & \left(\begin{array}{l} d; \mathit{coclfn}(b, \mathit{id}, \\ \quad \mathsf{if\ let}\ (c :: \mathit{fv}(() \rightarrow e_1)) = \mathit{id} \\ \quad\ \mathsf{then}\ e_1\ \mathsf{else}\ e'_{1,alt}, \\ \quad \mathsf{if\ let}\ (c :: x :: \mathit{fv}(x \rightarrow e'_2)) = \mathit{id} \\ \quad\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_{2,alt}); \\ (\mathsf{More},\ c :: \mathit{fv}(() \rightarrow e_1),\ c :: \mathit{fv}(x \rightarrow e'_2)) \end{array}\right) \\
 & & \quad \text{where} \quad c\ \text{fresh} \\
 & & \quad \text{and} \quad\ \ d; \mathit{coclfn}(b, \mathit{id}, e'_{1,alt}, e'_{2,alt}); e'_2 = \\
 & & \qquad\qquad\quad \mathit{defun}(d; \mathit{coclfn}(b, \mathit{id}, e_{1,alt}, e_{2,alt}); e_2) \\[1em]
\mathit{defun}\left(\begin{array}{l} d; \mathit{coclfn}(b, \mathit{id}, \\ \quad e_{1,alt}, e_{2,alt}); \\ x = e_0;\ e_1 \end{array}\right) & = & d; \mathit{coclfn}(b, \mathit{id}, e_{1,alt}, e_{2,alt}); x = e_0;\ \mathit{defun}(e_1) \\
\mathit{defun}\left(\begin{array}{l} d; \mathit{coclfn}(b, \mathit{id}, \\ \quad e_{1,alt}, e_{2,alt}); \\ e \end{array}\right) & = & d; \mathit{coclfn}(b, \mathit{id}, e_{1,alt}, e_{2,alt}); e \\
\mathit{coclfn}(b, \mathit{id}, e_{1,alt}, e_{2,alt}) & = & @\mathsf{cl}\ \mathsf{this.clfn} = \mathit{id} \rightarrow e_{1,alt}; \\
 & & @\mathsf{co}\ \mathsf{this.cofn} = \mathit{id} \rightarrow e_{2,alt}; b
\end{array}$$

Fig. 13. Defunctionalization.

Defunctionalization (Fig. 13). The *defun* function transforms the chains of let bindings and bind operators produced by *cps*, which contains three cases of syntax forms: (1) a bind operator ($e_1 \gg\!= e_2$), or (2) a let binding ($x = e_1;\ e_2$), or (3) the final expression. For (1), e_1 and e_2 are replaced by data structures that contain values for the free variables in e_1 and e_2 and are tagged with a fresh ID. The body of the expression is lifted to top-level synthetic definitions. For this, *defun* modifies the synthetic definitions b by extracting the body $e_{1,alt}$ of the synthetic clfn definition and the body $e_{2,alt}$ of cofn, and by adding an additional conditional clause to these definitions. The added clause answers to requests for a given ID with evaluating the original expression. For (2) and (3), *defun* recurses into the expressions.

After defunctionalization, lambdas $x \rightarrow e_0$ are lifted and assigned a top-level identifier id_0 and lambda applications, $id_0(e_1)$, are replaced with calls to a synthesized interpreter function $\mathsf{fn}(id_0, e_1)$. The latter branches on the identifier and executes the code that was lifted out of the original function.

Compiling. The *comp* function composes the compiler steps (not including *mnf*). We also define the *comp'* function, which jumps over the wrapping *trmp* expression and initialises the defunctionalisation with an environment that contains the two functions cofn and clfn, which assert false.

$$\begin{array}{lcl}
\mathit{comp} & = & \mathit{defun} \circ \mathit{cps} \circ \mathit{guard} \\
\mathit{comp}' & = & \mathit{defun}' \circ \mathit{cps}' \circ \mathit{guard}'
\end{array}$$

3.4 Semantics

We model the semantics as a reduction relation over configurations $p; q; c; m \ \rightarrow\ p'; q'; c'; m'$. Location $c = 0$ denotes contract execution, otherwise execution of client of address c. We distinguish

$$
\begin{array}{lllll}
(\textsc{Rgs}) & p; q; 0;\ \mathsf{awaitCl}_s(v, () \to e) & \to_g & p; q; 0;\ \mathsf{awaitCl}_t(c, () \to e) & \text{if } p; q; 0; v(c) \to^* p; q; 0; \mathsf{true} \\
(\textsc{Rbs}) & p; q; 0;\ \mathsf{awaitCl}_s(v, () \to e) & \to_b & p; q; 0;\ \mathsf{awaitCl}_t(c, () \to e) & \\
(\textsc{Rtm}) & p; q; 0;\ \mathsf{trmp}\left(\begin{array}{c}\mathsf{More},\\ v_1 :: e_1,\\ v_2 :: e_2\end{array}\right) & \to & p; q; 0; \left(\begin{array}{l} id\ =\ \mathsf{awaitCl}_t(c,\ \mathsf{this.clfn}(v_1 :: e_1));\\ \mathsf{trmp}(\mathsf{this.cofn}(v_2 :: id :: e_2))\end{array}\right) & \\
(\textsc{Rtd}) & p; q; 0;\ \mathsf{trmp}(\mathsf{Done},\ v) & \to & p; q; 0;\ v & \\
(\textsc{Rg}) & p; q; 0;\ \mathsf{awaitCl}_t(c, () \to e) & \to_g & p; q\ \mathsf{msg}(c, v)\ \mathsf{wr}(0, \mathsf{sender}, c); 0;\ v & \text{if } p; q; c; e \to^* p'; q'; c; v \\
(\textsc{Rb}) & p; q; 0;\ \mathsf{awaitCl}_t(c, () \to e) & \to_b & p; q\ \mathsf{msg}(c, v')\ \mathsf{wr}(0, \mathsf{sender}, c); 0;\ v' &
\end{array}
$$

Fig. 14. Evaluation (1/2).

good ($\to_g$) and bad ($\to_b$) evaluations (Fig. 14 and 15); shared rules are in black, without subscript ($\to$).

Attacker model. Attackers can control an arbitrary number of clients and make them send arbitrary messages. Hence, the bad semantics can answer a request to a client with an arbitrary message from an arbitrary *id*. We use evaluation with bad semantics to show that our compiler enforces access control against malicious clients.

Good evaluations of client expressions in the source language (Rgs) reduce to a client expression with a fixed client that fulfils the given predicate. We require that predicates evaluate purely. Hence, p and q do not change in the evaluation. On the other hand, bad evaluation of client expressions in the source language (Rbs) ignores the predicate, choosing an arbitrary client. Similarly, bad evaluation also chooses an arbitrary client for the evaluation of a trampoline in the target (Rtm), which does not specify a predicate. The trampoline ends when it reaches Done (Rtd). Further, after choosing a client to evaluate, the good evaluation (Rg) continues to reduce the client expression to a value, while the bad evaluation (Rb) replaces the expression e with a (manipulated) arbitrary value v'. Both evaluations (Rg, Rb) emit the message event $\mathsf{msg}(c, v)$ and an assignment to the special variable sender, when a client expressions is reduced to a value v, to record the client–contract interaction.

Common Evaluation (Fig. 15). Expressions are reduced under the evaluation context E on the current location (Re), assignment to variables is recorded in the trace (Rset°), accessing a variable is answered by the most recent assignment to it from the trace in the current location (Rget°). For synthetic variables, we use the synthetic store (Rget†, Rset†). Binary operators are defined as unsigned 256 bit integer arithmetic; we only show the rule for addition (Rop). Further, we give rules for conditionals (Rt, Rf), let binding (Rlet) and function application (Rlam) using pattern matching.

Pattern matching (Fig. 16). Matching $[x \Mapsto v]$ is a partial function, matching patterns x with values v, returning substitution of variables *id* to values. Matching is recursively defined over pairs; it matches constants to constants, identifiers to values by generating substitutions, and fails otherwise. Substitutions $[id \mapsto v]$, in turn, can be applied to terms e, written $[id \mapsto v]\ e$ (capture-avoiding substitution). Substitutions σ compose right-to-left $(\sigma\sigma')x = \sigma(\sigma' x)$.

3.5 Secure Compilation

We prove that the observable behavior of the contract before and after compilation is equivalent. We capture the observable behavior by execution traces and show that trace equivalence holds even when the program is attacked, i.e., reduced by $\to_b^*$.

Modelling Observable Behavior. The only source of observable nondeterminism in the bad semantics is the evaluation of $\mathsf{awaitCl}_s$ and $\mathsf{awaitCl}_t$. As clients decisions on message sending are influenced by the state of contract variables, tracking incoming client messages and state changes in

$$\begin{array}{llll}
(\textsc{Re}) & p;q;0;\ E[m] & \longrightarrow p';q';0;\ E[m'] & \text{if } p;q;0;m \longrightarrow p';q';0;m' \\
(\textsc{RGet}^{\circ}) & p;q;c;\ \mathsf{this}.i & \longrightarrow p;q;c;\ v & \text{if } \mathsf{wr}(c,i,v) \in p \\
(\textsc{RGet}^{\dagger}) & p;q;c;\ \mathsf{this}.j & \longrightarrow p;q;c;\ v & \text{if } \mathsf{wr}(c,j,v) \in q \\
(\textsc{RSet}^{\circ}) & p;q;c;\ \mathsf{this}.i := v & \longrightarrow p\ \mathsf{wr}(c,i,v);q;c;\ () & \\
(\textsc{RSet}^{\dagger}) & p;q;c;\ \mathsf{this}.j := v & \longrightarrow p;q\ \mathsf{wr}(c,j,v);c;\ () & \\
(\textsc{ROp}) & p;q;c;\ v_0\ +\ v_1 & \longrightarrow p;q;c;\ v' & \text{if } v'\ =\ v_0\ +\ v_1 \\
(\textsc{Rt}) & p;q;c;\left(\begin{array}{l}\mathsf{if\ let}\ x = v\\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1\end{array}\right) & \longrightarrow p;q;c;\ e_0' & \text{if } e_0'\ =\ [x \Mapsto v]\ e_0 \\
(\textsc{Rf}) & p;q;c;\left(\begin{array}{l}\mathsf{if\ let}\ x = v\\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1\end{array}\right) & \longrightarrow p;q;c;\ e_1 & \text{otherwise} \\
(\textsc{RApp}) & p;q;c;\ (x \longrightarrow e)\ v & \longrightarrow p;q;c;\ e' & \text{if } e'\ =\ [x \Mapsto v]\ e \\
(\textsc{RLet}) & p;q;c;\ (x = v;\ m) & \longrightarrow p;q;c;\ m' & \text{if } m'\ =\ [x \Mapsto v]\ m
\end{array}$$

Fig. 15. Evaluation (2/2).

$$\begin{array}{rl}
[c \Mapsto c] & = [] \\
[id \Mapsto v] & = [id \mapsto v] \\
[(e_0 :: e_1) \Mapsto (e_0' :: e_1')] & = [e_0 \Mapsto e_0'] \cdot [e_1 \Mapsto e_1']
\end{array}$$

Fig. 16. Pattern matching.

the trace suffices to capture the observable program behavior. If the observable behavior is the same for the source and the compiled programs, they are indistinguishable. Thus, behavior preservation amounts to trace equality on programs before and after compilation. Further, it suffices to model equality for non-stuck traces. The evaluation gets stuck (program crash) on assertions that guard against deviations from the intended program flow. The Ethereum Virtual Machine reverts contract calls that crash, i.e., state changes of crashed calls do not take effect, hence, stuck traces are not observable.

Since bad evaluation is nondeterministic, we work with not just programs, expressions and configurations, but program sets, expression sets, and configuration sets. Let $p;q;m \Downarrow$ be the trace set of the configuration $p;q;0;m$, e.g., the set of tuples of the final event sequence p' and value v of all reduction chains that start in $p;q;0;m$ and end in $p';0;q';v$. Our trace set definition does not include synthetic events q' of the final configuration. Synthetic events are introduced through compilation; excluding them allows us to put source and target trace sets in relation. Further, let the trace set of a configuration set $T \Downarrow$, be the union of the trace sets for each element:

$$p;q;m \Downarrow\ =\ \{\ (p',v) \mid (p;q;0;m) \longrightarrow_b^* (p';q';0;v)\ \}$$

$$T \Downarrow\ =\ \bigcup_{p;q;m \in T} p;q;m \Downarrow$$

We say that two configuration sets T and S are equivalent, denoted by $T \approx S$, iff T and S have the same traces sets:

$$(T \approx S)\ \Leftrightarrow\ (T \Downarrow\ =\ S \Downarrow)$$

By this definition, two expressions that eventually evaluate to the same value with the same trace are related by trace equality. We use this notion of trace equality to prove that a source program is trace-equal to its compiled version by evaluating the compiled program forward $\longrightarrow_b^*$ and the original program backward $\longleftarrow_b^*$ until configurations converge.

Secure Compilation. Theorem 1 states our correctness property, which says that observable traces generated by the malicious evaluation of programs are preserved ($\approx$) by compilation. The malicious evaluation models that client code has been replaced with arbitrary code, while contract code is unchanged. The preservation of observable traces implies the integrity of the (unchanged) contract code. Secure compilation guarantees that developers can write safe programs in the source language without knowledge about the compilation or the distributed execution of client/contract tiers.

Theorem 1 (Secure Compilation). *For each program P over closed terms, the trace set of the program under attack equals the trace set of the compiled program under attack:*
$\forall P.\ \{\ init_A(comp'(mnf'((P))))\ \} \approx \{\ init_A(P)\ \}.$

We first show that trace equality holds for the different compiler steps. Some compiler steps are defined as a recursive term-to-term transformation on open terms, whereas traceset equality is defined by reducing terms to values, i.e., on closed terms. Since all evaluable programs are closed terms, we show that the compiler steps preserve the traceset of an open term e that is closed by substitution $[x \Mapsto v]$. We formulate the necessary lemmas and sketch the proofs – the detailed proof is in Appendix C.

Lemma 1 (assoc correct). $\{\,p; q; [x{\Mapsto}v]\ assoc(m)\,\} \approx \{\,p; q; [x{\Mapsto}v]\ m\,\}$

Lemma 2 (mnf correct). $\{\,p; q; [x{\Mapsto}v]\ mnf(m)\,\} \approx \{\,p; q; [x{\Mapsto}v]\ m\,\}$

Lemma 3 (mnf' correct). $\{\,init_C(mnf'(d; b; m))\,\} \approx \{\,init_C(d; b; m)\,\}$

Lemma 4 (comp correct). $\{\,[x{\Mapsto}v]\ init_A(comp(d; b; \mathsf{trmp}(m)))\,\} \approx \{\,init_A(d; b; \mathsf{trmp}([x{\Mapsto}v]\ m))\,\}$

Lemma 5 (comp' correct). $\{\,[x{\Mapsto}v]\ init_A(comp'(d; b; \mathsf{trmp}(m)))\,\} \approx \{\,init_A(d; b; \mathsf{trmp}([x{\Mapsto}v]\ m))\,\}$

Proof sketch. Lemma 1–5 hold by chain of transitive trace equality relations. We show that a term is trace-equal to the same term after compilation, by evaluating the compiled program ($\rightarrow^*$) and the original program ($\leftarrow^*$) until configurations converge. In the inductive case, we can remove the current compiler step in redex position under traceset equality ($\approx$) since traces before and after applying the compiler step are equal by induction hypothesis.

An interesting case is the proof of *comp* for $P = d; b; \mathsf{awaitCl}(e_0, () \longrightarrow e_1)$. The compiler transforms the remote communication $\mathsf{awaitCl}_s$ into the use of a guard and a trampoline. The compiled program steps to the use of $\mathsf{awaitCl}_t$, the source program to $\mathsf{awaitCl}_s$. In the attacker relation $\rightarrow_b$, arbitrary clients can send arbitrary values with $\mathsf{awaitCl}_t$, leading to additional traces compared to the ones permitted in the source program where communication is modeled by $\mathsf{awaitCl}_s$. We observe that $\mathsf{awaitCl}_s$ generates the trace elements $\mathsf{msg}(c, v), \mathsf{wr}(0, \mathsf{sender}, c)$ for all v and that $\mathsf{awaitCl}_t$ generates the trace elements $\mathsf{msg}(c', v), \mathsf{wr}(0, \mathsf{sender}, c')$ for all v, c', which differ for $c' \neq c$.

Compilation adds an assert expression (Fig. 11) evaluated after receiving a value from a client. The assert gets stuck for configurations that produce trace elements with $c' \neq c$, removing the traces of such configurations from the trace set, leaving only the traces where $c' = c$. Hence, the trace set before and after compilation is equal under attack.

4 IMPLEMENTATION

Prisma is embedded into Scala (the host language) with its features implemented as a source-to-source macro expansion.[6]

[6] The implementation entails 21 Scala files, 3 412 lines of Scala source code (non-blank, non-comment) licensed under Apache 2.0 Open Source. The compiler phases are macros that recurse over the Scala AST: (a) the guarding phase, (b) the “simplifying” phase (including MNF translation, CPS translation of terms, defunctionalisation), and (c) the translation phase of (a subset) of Scala expressions and types to a custom intermediate representation based on Scala case classes. The intermediate representation is translated to Solidity code and passed to the Solidity compiler (solc).

The backend generating Solidity code is well separated. One could disable the compilation step to Solidity in the compilation pipeline, e.g., to run distributed code on multiple JVMs instead. In this case, the "contract code" would be executed by one computer (the "server"), and other computers would run the "client code".

The Scala runtime of Prisma contains the implementation of the serialisable datatypes, portable between Scala and the EVM (fixed-size arrays, dynamic arrays, unsigned integers of length of powers of two up to 256 bit). Our runtime wraps `web3j` [50] (for invoking transactions and interacting with the blockchain in general), `headlong` [72] (for serialisation/deserialisation in the Ethereum-specific serialisation format), as well as code to parse Solidity and Ethereum error messages and to translate them to Scala error messages.

5 EVALUATION

We evaluate Prisma along two research questions:

RQ1 *Does Prisma support the most common dApps scenarios?*
RQ2 *Do Prisma's abstractions affect performance?*

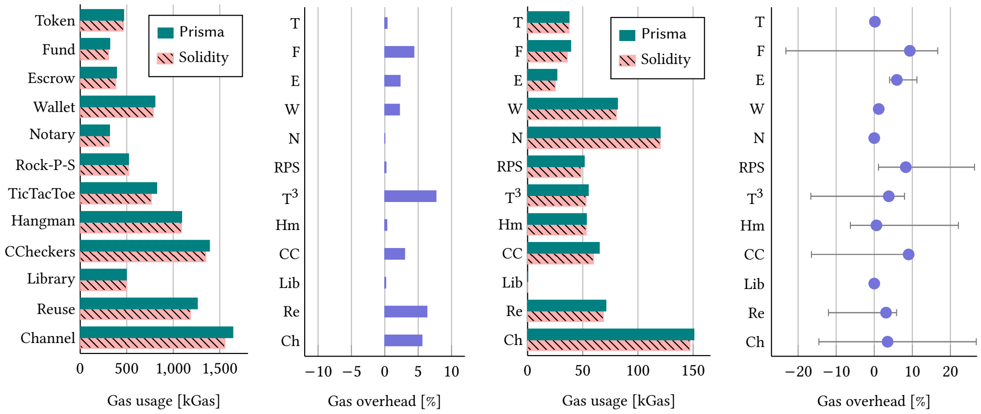
Case Studies and Expressiveness (RQ1). Five classes of smart contract applications have been identified [7]: Financial, Wallet, Notary, Game, and Library. To answer RQ1, we implemented at least one case study per category in Prisma. We implemented an ERC-20 Token,[7] a Crowdfunding, and an Escrowing dApp as representatives of financial dApps. We cover *wallets* by implementing a multi-signature wallet, a special type of wallet that provides a transaction voting mechanism by only executing transactions, which are signed by a fixed fraction of the set of owners. We implemented a general-purpose *notary* contract enabling users to store arbitrary data, e.g., document hashes or images, together with a submission timestamp and the data owner. As *games*, we implemented TicTacToe (Section 2), Rock-Paper-Scissors, Hangman and Chinese Checkers. Rock-Paper-Scissors makes use of timed commitments [3], i.e., all parties commit to a random seed share and open it after all commitments have been posted. The same technique can be used to generate randomness for dApps in a secure way. To reduce expensive code deployment, developers outsource commonly used logic to library contracts. We demonstrate *library*-based development in Prisma by including a TicTacToe library to our case studies and another TicTacToe dApp which uses that library instead of deploying the logic itself.

We also implemented a state channel [29, 30, 57] for TicTacToe in Prisma, which is an example for the class of *scalability solutions* that have emerged more recently. State channels enable parties to move parts of their dApp to a non-blockchain consensus system, falling-back to the blockchain in case of disputes, thereby making the dApps more efficient where possible.

Our case studies are between 1 K and 7.5 K bytes which is a representative size: Smart contracts are not built for large-scale applications since the gas model limits the maximal computation and storage volumes and causes huge fees for complex applications. The median (average, lower quantile, upper quantile) of the bytecode size of distinct contracts deployed at the time of writing is at 4 K (5.5 K, 1.5 K, 7.5 K) [44]. We further elaborate on the case studies including a comparison of the lines of code in Prisma compared to the equivalent lines in Solidity and Javascript in Appendix A. Our case studies demonstrate that Prisma supports most common dApps scenarios.

Performance of Prisma DApps (RQ2). Performance on the Ethereum blockchain is usually measured in terms of an Ethereum-specific metric called *gas*. Each instruction of the Ethereum Virtual Machine (EVM) consumes gas which needs to be paid for by the users in form of transaction fees credited to

[7] A study investigating all blocks mined until Sep 15, 2018 [62], found that 72.9 % of the high-activity contracts are token contracts compliant to ERC-20 or ERC-721, with an accumulated market capitalization of 12.7 B USD.



(a) Gas usage per deployment.

(b) Gas overhead per deployment.

(c) Gas usage per interaction.

(d) Gas overhead per interaction.

Fig. 17. The cost of abstraction. Gas overhead of contracts written with Prisma vs. Solidity. (The right plot displays minima, averages, maxima.)

the miner. We refer to the Ethereum yellow paper [91] for an overview of the gas consumption of the different EVM instructions. To answer RQ2, we implement our case studies in both Prisma and in Solidity/JavaScript and compare their gas consumption. Unlike prior work, we do not model a custom gas structure, but consider the real EVM gas costs [90].

Experimental setup. We execute each case study on different inputs to achieve different execution patterns that cover all contract functions. Each contract invocation that includes parameters with various sizes (e.g., dynamic length arrays) is executed with a range of realistic inputs, e.g., for Hangman, we consider several words (2 to 40 characters) and different order of guesses, covering games in which the guesser wins and those in which they lose. Prisma and Solidity/JavaScript implementations are executed on the same inputs.

We perform the measurements on a local setup. As the execution in the Ethereum VM is deterministic, a single measurement suffices. We set up the local Ethereum blockchain with *Ganache* (Core v2.13.2) on the latest supported hard fork (Muir Glacier). All contracts are compiled to EVM byte code with *solc* (v0.8.1, optimized on 20 runs). We differentiate contract deployment and contract interaction. Deployment means uploading the contract to the blockchain and initializing its state, which occurs just once per application instance. A single instance typically involves several contract interactions, i.e., transactions calling public contract functions.

Results. Fig. 17 shows the average gas consumption of contract deployment (Fig. 17a) and interaction (Fig. 17c) as well as the relative overhead of Prisma vs. Solidity/JS of deployment (Fig. 17b) and interaction (Fig. 17d). As the gas consumption of contract invocations depends heavily on the executed function, the contract state, and the inputs, we provide the maximal, minimal and averaged overhead. The results show that the average gas consumption of Prisma is close to the one of Solidity/JS. Our compiler achieves a deployment overhead of maximally 6 % (TicTacToe) or 86 K gas (TicTacToe Channel). The interaction overhead is below 10 % for all case studies which at most amounts to 3.55 K gas.[8]

Prisma's deployment overhead is mainly due to the automated flow control. To guarantee correct execution, Prisma manages a state variable for dApps with more than one state. The storage reserved for and the code deployed to maintain the state variable cause a constant cost of around

[8] equals 0.59 USD based on gas price and exchange course of April 15, 2021

Table 18. Related work.

Language	Encoding	Perspective	Protocol
Solidity	FSM	Local	Assertions
Obsidian	FSM	Local	Type states
Nomos	MNF	Local	Session types
Prisma	DS	Global	Control flow

45 K gas. In Solidity, developers manually check whether flow control is needed and, if so, may derive the state from existing contract variables to avoid a state variable if possible.

The Token, Notary, Wallet and Library case studies do not require flow control: each function can be called by any client at any time. Hence, their overhead is small. Escrow, Hangman and Rock-Paper-Scissors require a state variable, also in Solidity – which partially compensates the overhead of Prisma's automated flow control. Crowdfunding, Chinese Checkers, TicTacToe (Library and Channel) do not require an explicit state variable in Solidity, as the state can be derived from the contract variables, e.g., the number of moves. Thus, these case studies have the largest deployment overhead.

While the average relative interaction overhead is constantly below 10 %, some contract invocations are far above, e.g., in Crowdfunding, TicTacToe Channel, and Rock-Paper-Scissors. Yet, case studies with such spikes also involve interactions that are executed within the same dApp instance with a negative overhead and amortize the costs of more costly transactions. These deviations are also mainly due to automated flow control. In EVM, setting a zero variable to some non-zero value costs more gas (20 K gas) than changing its value (5 K gas) [90], and setting the value to zero saves gas. Occupying and releasing storage via the state variable can cost or save gas in a different way than in traditional dApps without an explicit state variable, leading to different (and even negative) overhead in different transactions.

Besides the gas-overhead, we also consider the time-overhead of Prisma. In Ethereum, the estimated confirmation time for transactions is 3-5 minutes (assuming no congestion), which makes the number of on-chain interactions dominate the total execution time. As Prisma preserves the number of on-chain interactions, we assess the time-overhead of Prisma, if any, to be negligible.

Note that per se it is not possible to achieve a better gas consumption in Prisma than in Solidity – every contract compiled from Prisma can be implemented in Solidity. Given the abstractions we offer beyond the traditional development approach, and the sensibility of smart contracts to small changes in instructions, we conclude that our abstractions come with acceptable overhead. We are confident that further engineering effort can eliminate the observed overhead.

Threats to validity. The main threat is that the manually written code may be optimized better or worse than the code generated by the compiler. We mitigate this threat by applying all gas optimizations, our compiler performs automatically, to the Solidity implementations. An external threat is that changes in the gas pricing of Ethereum may affect our evaluation. For reproducibility, we state the Ethereum version (hard fork), we used in the paper.

6 DISCUSSION AND RELATED WORK

6.1 Smart Contract Languages for Enforcing Protocols

We compare Prisma to Solidity, Obsidian [18–20], and Nomos [25, 26]. We highlight these languages as those also address the correctness of the client–contract interactions. Tab. 18 overviews the features of the surveyed languages for (a) the *perspective* of defining interacting parties, (b) the used *encoding* of the interaction effects, and (c) the method used to check the contract-client interaction

```
asset contract TTT {
  state Funding{}; state Executing{}; state Finished{}; state Closed{}
  transaction Fund(TTT@Funding>>(Funding|Executing) this, int c) {
    /*...*/;  if (/* enough funds? */) -> Executing else -> Funding }
  transaction Move(TTT@Executing>>(Executing|Finished) this, int x, int y) {
    /*...*/; if (/* game over? */) -> Finished else -> Executing }
  transaction Payout(TTT@Finished>>Closed this) {
    /*...*/; -> Closed } }
```

Fig. 19. Obsidian.

```
type Funding   =       int -> +{ notenough: Funding, enough: Executing }
type Executing = int -> int -> +{ notdone: Executing, done: Finished }
type Finished  =       int -> 1
proc contract funding : . |{*}- ($s : Funding) = {
  a = recv $s; /* ... */
  if /* enough funds? */ then  $s.notenough; $s <- funding
                         else  $s.enough; $s <- executing }
proc contract executing : . |{*}- ($s : Executing) = {
  x = recv $s; y = recv $s; /* ... */
  if /* game over? */ then $s.notdone; $s <- executing
                      else $s.done; z = recv $s; close $s }
```

Fig. 20. Nomos.

$$\textsc{NomosR}\quad \frac{\Psi; \Gamma, (y{:}A) \vdash P :: (c : B)}{\Psi; \Gamma \vdash (y \leftarrow \text{recv } c;\ P) :: (c : A \multimap B)} \qquad \textsc{NomosS}\quad \frac{\Psi; \Gamma \vdash P :: (c : B)}{\Psi; \Gamma, (w{:}A) \vdash (\text{send } c\ w;\ P) :: (c : A \otimes B)}$$

$$\textsc{Obsidian}\quad \frac{(\text{transaction } T\ m(\overline{t.(s \gg s')\ x})\{\ldots\}) \in \text{members}_{t_0}}{\Delta, \overline{e{:}t.s} \vdash e_0.m(\overline{e}) : T \dashv \Delta, \overline{e{:}t.s'}}$$

Fig. 21. Excerpts of simplified Nomos and Obsidian typing rules.

protocol. Fig. 4c, 19, and 20 show code snippets in these languages, each encoding the *TicTacToe* state machine from Fig. 1. All three languages focus solely on the contract and do not state how clients are developed, hence only contract code is shown.

All three approaches take a **local perspective** on interacting parties: Contract and clients are defined separately, and their interaction is encoded by explicit send and receive side effects. In Solidity and Obsidian, receive corresponds to arguments and send to return values of methods defined in the contract classes. In Nomos, send and receive are expressed as procedures operating over a channel – given a channel `c`, sending and receiving is represented by explicit statements (`x = recv c; ...` and `send c x; ...`).

The approaches differ in the **encoding style** of communication effects. Solidity and Obsidian adopt an *FSM-style encoding*: Contract fields encode states, methods encode transitions. The contract in Fig. 4c represents FSM states via the `phase` field with initial state `Funding` (Line 30). The `Fund`, `Move` and `Payout` methods are transitions, e.g., `Payout` transitions the contract into the final state `Closed` (Line 51). The FSM-style encoding results in an implicitly-everywhere concurrent programming model, which is complex to reason about and unfitting for dApps because the execution model of blockchains is inherently sequential – all method invocations are brought into a world-wide total order. Nomos adopts the *monadic normal form (MNF)* via do-notation to order effects. While the implementation of TicTacToe in FSM style requires three methods(`Fund`, `Move`, `Payout` – one per transition), we only need two methods in MNF-style (`funding`, `executing` – one per state with multiple entry points), and a single method in DS-style (`init`). For instance, the sequence of states and transitions $\mathit{Executing} \xrightarrow{\mathit{Move}(x,y)} \mathit{Finished} \xrightarrow{\mathit{Payout}()} \mathit{Closed}$ in Nomos can be written sequentially in do-notation by inlining the last function which only has a single entry point. Still, do-notation can be cumbersome (e.g., funding and executing in Nomos are separate methods that cannot be inlined since they have multiple entry points and model loops).

All three languages require an **explicit protocol** for governing the send–receive interactions, to ensure that every send effect has a corresponding receive effect in an interacting – separately defined – party. In Solidity, developers express the protocol via run-time assertions to guard against invoking the methods in an incorrect order (e.g., `require(phase==Finished)` in Fig. 4c, Line 40). Unlike Solidity, which does not support statically checking protocol compliance, Nomos and Obsidian employ *behavioral typing* for static checks. Deployed contracts may interact with third-party, potentially manipulated clients. Compile-time checking alone cannot provide security guarantees. Yet, complementing run-time enforcement with static checks helps detecting cases that are guaranteed to fail at run time ahead of time.

Obsidian. Obsidian employs typestates to increase safety of contract–client communication. Contracts define a number of typestates; A method call can change the typestate of an object, and calling a method on a receiver that is in the wrong typestate results in a typing error. Each method in Fig. 19 is annotated with the state in which it can be called, e.g., `Payout` requires state `Finished`, and transitions to `Closed` (Line 7).

Nomos. Nomos employs session types. The session types `Funding`, `Executing`, `Finished` in Fig. 20 encode the protocol. Receiving a message is represented by a function type, e.g., in the `Funding` state, we receive an integer `int -> ...` (Line 1). We respond by either repeating the funding (`Funding`), or continuing to the next state of the protocol (`Executing`). This is represented by internal choice `+{ ... }` that takes multiple possible responses giving each of them a unique label (`notenough` and `enough`). Type `1` indicates the end of a protocol (Line 3). The contract processes `funding` (Line 4) and `executing` (Line 8) implement the protocol. The `recv` operation (Line 5) takes a session-typed channel of form `T -> U`, returns a value of type `T` and changes the type of the channel to `U`. A session type for internal choice (`+{ ... }`), requires the program to select one of the offered labels (e.g., `$s.notenough` in Line 6 and `$s.enough` in Line 7), e.g., in the left and right branch of a conditional statement.

Type systems. We show excerpts of simplified typing rules for Nomos and Obsidian (Fig. 21). Nomos rules have the form $\Psi; \Gamma \vdash P :: (c{:}A)$. A process P offers a channel c of type A with values in context Ψ and channels in Γ. We can see that variables change their type to model the linearity of session types in the NomosS (and NomosR) rule: Sending (and receiving) changes the type of the channel c from $A \multimap B$ to B (and $A \otimes B$ to B). Obsidian rules have the form $\Delta \vdash e{:}t \dashv \Delta'$. An expression e has type t in context Δ and changes Δ to Δ'. We can see that variables change their type on method invocation (Obsidian): A method m in class t_0 with arguments e_i of type t_i, returning T, changes the type state of the arguments from s_i to s_i'. For Prisma, instead, a standard judgement $\Gamma \vdash e : T$ suffices for communication. Variables do not change their type. $\mathsf{awaitCl}(p)\{b\}$ has type T in context Γ if p is a predicate of Addr and b is a pair of Ether and T:

$$\textsc{Prisma}\quad \frac{\Gamma \vdash p : \mathit{Addr} \rightarrow \mathit{Bool} \qquad \Gamma \vdash b : \mathit{Ether} \times T}{\Gamma \vdash \mathsf{awaitCl}(p)\{\ b\ \} : T}$$

Prisma. As shown in Tab. 18, Prisma occupies an unexplored point in the design space: *global* instead of local perspective on interacting parties, *direct style (DS)* instead of FSM or MNF encoding of effects, and *control flow* instead of extra protocol for governing interactions.

Prisma takes a **global perspective** on interacting parties. The parties execute the same program, where 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. Hence, dApps are modeled as sequences and loops of send-receive-instructions shared by interacting parties. Due to the global direct style perspective, it is syntactically impossible to define parties with mismatching send and receive pairs. Hence, a standard System-F-like type system suffices. The interaction protocol follows directly from the sequential **control flow** of interaction points in the program – the compiler can automatically generate access and control guards with correctness guarantees. Semantically, Prisma features a by-default-sequential programming model, intentionally making the sequential execution of methods explicit, including interaction effects.

The global direct-style model also leads to improved design of dApps: No programmatic state management on the contract and no so-called *callback hell* [31] on the client. The direct style is also superior to Nomos' MNF style. The tierless model avoids boilerplate: Client code can directly access public contract variables, unlike JavaScript code, which has to access them via a function call that requires either an await expression or a callback.[9] Additionally, the developer has to implement getters for public variables with complex data types such as arrays.[10] We provide some code measurements (lines of code and number of cross-tier control-flow calls) of our Prisma and Solidity/JS dApp case studies in Appendix B.

Finally, using one language for both the contract and the clients naturally enables static type safety of values that cross the contract–client boundary: an honest, non-compromized client cannot provide inconsistent input, e.g., with wrong number of parameters or falsely encoded types.[11] In a setting with different language stacks, it is not possible to statically detect type mismatches in the client–contract interaction; e.g., Solidity has a type *bytes* for byte arrays, which does not exist in JavaScript (commonly used to implement clients of a Solidity contract). Client developers need to encode byte arrays using hexadecimal string representations starting with "0x", otherwise they cannot be interpreted by the contract.

6.2 Other Related Work

Smart contract languages. Harz and Knottenbelt [45] survey smart contract languages, Hu et al. [48] survey smart contract tools and systems, Wöhrer and Zdun [89] give an overview of design patterns in smart contracts. Brünjes and Gabbay [13] distinguish between imperative and functional smart contract programming. *Imperative contracts* are based on the account model; the most prominent language is Solidity [32]. *Functional* ones [14, 77, 78] are based on EUTxO (Extended Unspent Transaction Output) model [39]. State channels [15, 29, 30, 57] optimistically optimize contracts for the functional model. Prisma does not yet support compilation to state channels but we plan to treat them as another kind of tier.

[9] Obsidian and Nomos do not provide any client design, so we can only compare to Solidity/JavaScript.

[10] For simple data types the getter is generated automatically.

[11] Recall that in dApps checking cross-tier type-safety is not a security feature but a design-time safety feature (due to the open-world assumption of the execution model of public ledgers).

Smart contracts as state machines. Scilla [79] is an automata-based compiler for contracts. FSolidM [55] enables creating contracts via a graphical interface. VeriSolid [56] generates contracts from graphical models enriched with predicates based on computational tree logic. EFSM tools [84] generate contracts from state machines and linear temporal logic. Prisma avoids a separate specification but infers transactions and their order from the control flow of a multitier dApp.

Analysis tools. Durieux et al. [28] and Ferreira et al. [35] empirically validate languages and tools and relate design patterns to security vulnerabilities, extending the survey by Di Angelo and Salzer [4]. Our work is complementary, targeting the correctness of the distributed program flow. For vulnerabilities not related to program flow (e.g., front-running, or bad randomness), developers (using Solidity/JavaScript or Prisma) can use the surveyed analysis tools.

Multitier languages. Multitier programming was pioneered by Hop [80, 81]. Modeling a persistent session in client–server applications with continuations was mentioned by Queinnec [69] and elaborated in Links [22, 38]. Eliom [70] supports bidirectional client–server communication for web applications. ScalaLoci [87] generalizes the multitier model to generic distributed architectures. Our work specializes it to the dApp domain and its specific properties. Giallorenzo et al. [41] establish interesting connections between multitier (subjective) and choreographic (objective) languages – two variants of the global model. Prisma adopts the subjective view, which naturally fits the dApp domain, where a dominant role (contract) controls the execution and diverts control to other parties (clients) to collect their input.

Mashic [51] is a compiler for a *mashup* between two JavaScript programs: the untrusted embedded (iframe) *gadget(s)* and the trustworthy hosting *integrator* program, which communicate via messages. The authors prove that the compiler guarantees integrity and confidentiality. More specifically, the gadget(s) cannot learn more than what the integrator sends and, analogously, the gadget's influence is limited to the integrators interface. In Mashic, the two programs are separate and the compiler checks that they communicate only via specified messages. In contrast, in Prisma, client and contract code are mixed. Thus, in addition to checking that only the specified messages are used, we can also check the interaction protocol – expressed by the structure of the control flow of the program – and ensure that it is followed by the target program after compilation.

Swift's [17] secure automatic partitioning approach uses information flow policies to derive placements. Based on the policies, a constraint solver with integer programming heuristically picks a placement such that network traffic is minimal and information flow integrity is preserved. In contrast, placements in Prisma are explicit to the developer. Further, in blockchain programming, every single instruction generated by the compiler potentially incurs high costs. Therefore, we demonstrated that our compiler generates inexpensive programs, whereas Swift does not consider the program's execution cost.

Effectful programs and meta-programming. MNF and CSP are widely discussed as intermediate compiler forms [6, 21, 37, 49, 54]. F# computation expressions [64] support control-flow operators in monadic expressions. OCaml supports a monadic and applicative let [88]: more flexible than do-notation but still restricted to MNF. Idris' !-notation [10] inspired the GHC proposal for monadic inline binding [68]. Scala supports effectful programs through coroutines [67], async/await [73], monadic inline binding [11], Dsl.scala [92] and a (deprecated) compiler plugin for CPS translation [74]. The dotty-cps-async macro [82] supports async/await and similar effects for the Dotty compiler.

7 CONCLUSION

We proposed Prisma, the first global language for dApps that features direct style communication. Compared to the state of the art, Prisma (a) enables the implementation of contract and client logic within the same development unit, rendering intricacies of the heterogeneous technology stack obsolete and avoiding boilerplate code, (b) provides support for explicitly encoding the intended program flow and (c) reduces the risk of human failures by enforcing the intended program flow and forcing developers to specify access control.

Unlike previous work that targeted challenges in the development of dApps with advanced typing disciplines e.g., session types, our model does not exhibit visible side effects and gets away with a standard System-F-style type system. We describe the design and the main features of Prisma informally, define its formal semantics, formalize the compilation process and prove it correct. We demonstrate Prisma's applicability via case studies and performance benchmarks.

We plan to generate state channels – to optimistically cost-optimize dApps – similar to how we generate state machines from high-level logic. Further, we believe that our technique for deriving the communication protocol from direct-style control flow generalizes beyond the domain of smart contracts and we will explore its further applicability.

ACKNOWLEDGMENTS

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).

Thanks to George Zakhour for feedback on the initial draft.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (2009), 4:1–4:40. https://doi.org/10.1145/1609956.1609960

[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* IEEE Computer Society, 443–458. https://doi.org/10.1109/SP.2014.35

[4] Monika Di Angelo and Gernot Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019, Newark, CA, USA, April 4-9, 2019.* IEEE, 69–78. https://doi.org/10.1109/DAPPCON.2019.00018

[5] Monika Di Angelo and Gernot Salzer. 2020. Wallet Contracts on Ethereum. *CoRR* abs/2001.06909 (2020). arXiv:2001.06909 https://arxiv.org/abs/2001.06909

[6] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

[7] Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security.* Springer, 494–509.

[8] Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C#. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 233–257. https://doi.org/10.1007/978-3-642-31057-7_12

[9] Sam Blackshear, Evan Cheng, D. Dill, Victor Gao, B. Maurer, T. Nowacki, Alistair Pott, S. Qadeer, Dario Russi, Stephane Sezer, Tim Zakian, and Run tian Zhou. 2019. Move: A Language With Programmable Resources. https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf.

[10] Edwin Brady. 2007. The Idris Tutorial. Interfaces. Monads and do-notation. !-notation. http://docs.idris-lang.org/en/latest/tutorial/interfaces.html#notation. Accessed 14-11-2020.

[11] Flavio W. Brasil and Sameer Brenn. 2017. Monadless – Syntactic sugar for monad composition in Scala. https://github.com/monadless/monadless.

[12] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2016. An In-Depth Look at the Parity Multisig Bug. hackingdistributed.com/2017/07/22/deep-dive-parity-bug/. Accessed 14-11-2020.

[13] Lars Brünjes and Murdoch James Gabbay. 2020. UTxO- vs Account-Based Smart Contract Blockchain Programming Paradigms. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12478)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 73–88. https://doi.org/10.1007/978-3-030-61467-6_6

[14] Manuel Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompson, Rebecca Valentine, and Philip Wadler. 2019. Functional Blockchain Contracts. (2019). https://iohk.io/en/research/library/papers/functional-blockchain-contracts/

[15] Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. 2020. Hydra: Fast Isomorphic State Channels. *IACR Cryptol. ePrint Arch.* 2020 (2020), 299. https://eprint.iacr.org/2020/299

[16] Kwanghoon Choi and Byeong-Mo Chang. 2019. A Theory of RPC Calculi for Client–Server Model. *Journal of Functional Programming* 29 (2019). https://doi.org/10.1017/S0956796819000029

[17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure web applications via automatic partitioning. In *Symposium on Operating Systems Principles*.

[18] Michael J. Coblenz. 2017. Obsidian: a safer blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 97–99. https://doi.org/10.1109/ICSE-C.2017.150

[19] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Jonathan Aldrich, Joshua Sunshine, and Brad A. Myers. 2019. User-Centered Programming Language Design in the Obsidian Smart Contract Language. *CoRR* abs/1912.04719 (2019). arXiv:1912.04719 http://arxiv.org/abs/1912.04719

[20] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. *CoRR* abs/1909.03523 (2019). arXiv:1909.03523 http://arxiv.org/abs/1909.03523

[21] Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.* 3, ICFP (2019), 79:1–79:28. https://doi.org/10.1145/3341643

[22] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO'06)*. Springer-Verlag, Berlin, Heidelberg, 266–296. http://dl.acm.org/citation.cfm?id=1777707.1777724

[23] Phil Daian. 2016. Analysis of the DAO exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. Accessed 14-11-2020.

[24] Dapp.com. 2020. 2020 Q2 Dapp Market Report. https://www.dapp.com/article/q2-2020-dapp-market-report.

[25] Ankush Das, S. Balzer, J. Hoffmann, and F. Pfenning. 2019. Resource-Aware Session Types for Digital Contracts. *ArXiv* abs/1902.06056 (2019).

[26] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2021. Nomos: A Protocol-Enforcing, Asset-Tracking, and Gas-Aware Language for Smart Contracts. (2021).

[27] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. 2009. Sessions and Session Types: An Overview. In *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6194)*, Cosimo Laneve and Jianwen Su (Eds.). Springer, 1–28. https://doi.org/10.1007/978-3-642-14458-5_1

[28] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 530–541. https://doi.org/10.1145/3377811.3380364

[29] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, 625–656. https://doi.org/10.1007/978-3-030-17653-2_21

[30] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 949–966.

https://doi.org/10.1145/3243734.3243856

[31] Jonathan Edwards. 2009. Coherent reaction. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 925–932. https://doi.org/10.1145/1639950.1640058

[32] Ethereum Foundation. 2015. Solidity Documentation. https://docs.soliditylang.org/en/v0.8.1/. Accessed 14-11-2020.

[33] Ethereum Foundation. 2015. Solidity Documentation – Common Patterns. https://docs.soliditylang.org/en/v0.7.4/common-patterns.html. Accessed 14-11-2020.

[34] Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

[35] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. *CoRR* abs/2007.04771 (2020). arXiv:2007.04771 https://arxiv.org/abs/2007.04771

[36] Klint Finley. 2016. A $50 Million Hack Just Showed That the DAO Was All Too Human. *Wired* (6 2016).

[37] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

[38] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 28 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290341

[39] Murdoch James Gabbay. 2020. What is an EUTxO blockchain? *CoRR* abs/2007.12404 (2020). arXiv:2007.12404 https://arxiv.org/abs/2007.12404

[40] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Choreographies as Objects. arXiv:2005.09520 [cs.PL]

[41] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty Languages: The Choreographic and Multitier Cases (Pearl). In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:27. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22

[42] PolyCrypt GmbH. 2020. Perun Network. https://perun.network.

[43] Google Inc. 2021. Google Cloud BigQuery: Contract deployment per month. https://console.cloud.google.com/bigquery. Query: SELECT EXTRACT(MONTH FROM c.block_timestamp) AS m, EXTRACT(YEAR FROM c.block_timestamp) AS y, COUNT(c.address) FROM 'bigquery-public-data.ethereum_blockchain.live_contracts' AS c GROUP BY m, y ORDER BY y, m; Accessed 07-07-2021.

[44] Google Inc. 2021. Google Cloud BigQuery: Contract size. https://console.cloud.google.com/bigquery. Query: WITH d as (SELECT DISTINCT c.bytecode,(LENGTH(c.bytecode)-2)/2 as s FROM '[...]live_contracts' AS c) SELECT PERCENTILE_CONT(d.s, [0, 0.25, 0.5, 0.75, 1]) OVER () AS M FROM d LIMIT 1; Accessed 14-11-2021.

[45] Dominik Harz and William J. Knottenbelt. 2018. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. *CoRR* abs/1809.09805 (2018). arXiv:1809.09805 http://arxiv.org/abs/1809.09805

[46] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. https://doi.org/10.1145/359576.359585

[47] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6536)*, Raja Natarajan and Adegboyega K. Ojo (Eds.). Springer, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4

[48] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. 2020. A Comprehensive Survey on Smart Contract Construction and Execution: Paradigms, Tools and Systems. *CoRR* abs/2008.13413 (2020). arXiv:2008.13413 https://arxiv.org/abs/2008.13413

[49] Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 177–190. https://doi.org/10.1145/1291151.1291179

[50] Web3 Labs. 2016. Web3j: Web3 Java Ethereum Ðapp API (GitHub Repository). https://github.com/web3j/web3j.

[51] Zhengqin Luo and Tamara Rezk. 2012. Mashic Compiler: Mashup Sandboxing Based on Inter-frame Communication. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, Stephen Chong (Ed.). IEEE Computer Society, 157–170. https://doi.org/10.1109/CSF.2012.22

[52] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[53] Simon Marlow. 2010. Haskell 2010: Language Report. Expressions. Do Expressions. https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-470003.14. Accessed 14-11-2020.

[54] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. https://doi.org/10.1145/3062341.3062380

[55] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10957)*, Sarah Meiklejohn and Kazue Sako (Eds.). Springer, 523–540. https://doi.org/10.1007/978-3-662-58387-6_28

[56] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 446–465. https://doi.org/10.1007/978-3-030-32101-7_27

[57] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. 2019. Sprites and State Channels: Payment Networks that Go Faster Than Lightning. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 508–526. https://doi.org/10.1007/978-3-030-32101-7_30

[58] Mix. 2019. These are the top 10 programming languages in blockchain. https://thenextweb.com/hardfork/2019/05/24/javascript-programming-java-cryptocurrency/. Accessed 14-11-2020.

[59] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2014. Service-Oriented Programming with Jolie. In *Web Services Foundations*, Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel (Eds.). Springer, 81–107. https://doi.org/10.1007/978-1-4614-7518-7_4

[60] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[61] Reed Oei, Michael J. Coblenz, and Jonathan Aldrich. 2020. Psamathe: A DSL with Flows for Safe Blockchain Assets. *CoRR* abs/2010.04800 (2020). arXiv:2010.04800 https://arxiv.org/abs/2010.04800

[62] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming Jack Jiang. 2020. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering* 25 (2020), 1864–1904. Issue 3. https://doi.org/10.1007/s10664-019-09796-5

[63] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (2019), 125:1–125:36. https://doi.org/10.1145/3280984

[64] Tomas Petricek and Don Syme. 2014. The F# Computation Expression Zoo. In *PADL (Lecture Notes in Computer Science, Vol. 8324)*. Springer, 33–48.

[65] Sergey Petrov. 2017. Another Parity Wallet hack explained. https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c

[66] Andrew M. Pitts. 2000. Operational Semantics and Program Equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures (Lecture Notes in Computer Science, Vol. 2395)*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.). Springer, 378–412. https://doi.org/10.1007/3-540-45699-6_8

[67] Aleksandar Prokopec. 2015. Scala Coroutines. https://github.com/storm-enroute/coroutines.

[68] Jon Purdy. 2017. Discussion on GHC Pre-Proposal: Add InlineBindings proposal. https://github.com/ghc-proposals/ghc-proposals/pull/64. Accessed 14-11-2020.

[69] Christian Queinnec. 2000. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 23–33. https://doi.org/10.1145/351240.351243

[70] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A core ML language for Tierless Web Programming. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems* (Hanoi, Vietnam) *(APLAS '16)*, Atsushi Igarashi (Ed.). Springer-Verlag, Berlin, Heidelberg, 377–397. https://doi.org/10.1007/978-3-319-47958-3_20

[71] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM '72*.

[72] Evan Saulpaugh. 2018. Headlong (GitHub Repository). https://github.com/esaulpaugh/headlong.

[73] Scala Development Team. 2012. scala-async. A Scala DSL to enable a direct style of coding when composing Futures. https://github.com/scala/scala-async.

[74] Scala Development Team. 2013. scala-continuations. The Scala delimited continuations plugin and library. https://github.com/scala/scala-continuations.

[75] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, Stefan Marr and Jennifer B. Sartor (Eds.). ACM, 218–219. https://doi.org/10.1145/3191697.3213790

[76] Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. 2019. Flint for Safer Smart Contracts. *CoRR* abs/1904.06534 (2019). arXiv:1904.06534 http://arxiv.org/abs/1904.06534

[77] Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon THompson. 2020. Marlowe: implementing and analysing financial contracts on blockchain, Tiziana Margaria and Bernhard Steffen (Eds.). *Workshop on Trusted Smart Contracts @ FC 2020*. https://iohk.io/en/research/library/papers/marloweimplementing-and-analysing-financial-contracts-on-blockchain/

[78] Pablo Lamela Seijas and Simon J. Thompson. 2018. Marlowe: Financial Contracts on Blockchain. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 11247)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 356–375. https://doi.org/10.1007/978-3-030-03427-6_27

[79] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *PACMPL* 3, OOPSLA (2019), 185:1–185:30. https://doi.org/10.1145/3360611

[80] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop, A Language for Programming the Web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, USA) *(OOPSLA Companion '06)*. ACM, New York, NY, USA.

[81] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP '16)*. ACM, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

[82] Ruslan Shevchenko. 2020. dotty-cps-async. https://github.com/rssh/dotty-cps-async.

[83] State Channels contributors. 2020. State Channels. https://statechannels.org/.

[84] Dmitrii Suvorov and Vladimir Ulyantsev. 2019. Smart Contract Design Meets State Machine Synthesis: Case Studies. *CoRR* abs/1906.02906 (2019). arXiv:1906.02906 http://arxiv.org/abs/1906.02906

[85] Uniswap Labs. 2021. Uniswap Info. https://v2.info.uniswap.org/home. Accessed 07-07-2021.

[86] Philip Wadler. 2012. Propositions as sessions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 273–286. https://doi.org/10.1145/2364527.2364568

[87] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[88] Leo White. 2018. OCaml: Add "monadic" let operators. https://github.com/ocaml/ocaml/pull/1947.

[89] Maximilian Wöhrer and Uwe Zdun. 2020. From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns. *IEEE Softw.* 37, 4 (2020), 37–42. https://doi.org/10.1109/MS.2020.2993470

[90] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Ethereum Yellow Paper: A Formal Specification of Ethereum, a Programmable Blockchain. BERLIN VERSION 0e0eba8 – 2021-11-02. Accessed 14-11-2020.

[91] Gavin Wood. 2022. Ethereum Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf.

[92] Bo Yang. 2016. Dsl.scala – A framework to create embedded Domain-Specific Languages in Scala. https://github.com/ThoughtWorksInc/Dsl.scala.

A CASE STUDIES

This section describes the implemented case studies in detail. Bartoletti and Pompianu [7] identify five classes of smart contract applications: Financial, Notary, Game, Wallet, and Library. Our case studies include at least one application per category (Table 22). In addition, we consider scalability solutions.

Financial. These apps include digital tokens, crowdfunding, escrowing, advertisement, insurances and sometimes Ponzi schemes. A study investigating all blocks mined until September 15th, 2018 [62], found that 72.9 % of the high-activity contracts are token contracts compliant to ERC-20 or ERC-721, which have an accumulated market capitalization of US $ 12.7 billion. We have implemented a fungible Prisma token of the ERC-20 standard. Further, we implemented crowdfunding and escrowing case studies. These case studies demonstrate how to send and receive coins with Prisma, which is the basic functionality of financial applications. Other financial use cases can be implemented in Prisma with similar techniques.

Notary. These contracts use the blockchain to store data immutably and persistently, e.g., to certify their ownership. We implemented a general-purpose notary contract enabling users to store arbitrary data, e.g., document hashes or images, together with a submission timestamp and the data owner. This case study demonstrates that Notaries are expressible with Prisma.

Games. We implemented TicTacToe (Section 2), Rock-Paper-Scissors, Hangman and Chinese Checkers. Hangman evolves through multiple phases and hence benefits from the explicit control flow definition in Prisma more than the other game case studies. The game Chinese Checkers is more complex than the others, in regard to the number of parties, the game logic and the number of rounds, and hence, represents larger applications. Rock-Paper-Scissors illustrates how randomness for dApps is securely generated. Every Ethereum transaction, including the executions of contracts, is deterministic – all participants can validate the generation of new blocks. Hence, secure randomness is negotiated among parties: in this case, by making use of timed commitments [3], i.e., all parties commit to a random seed share and open it after all commitments have been posted. The contract uses the sum of all seed shares as randomness. If one party aborts prior to opening its commitment, it is penalized. In Rock-Paper-Scissors both parties commit to their choice – their random share – and open it afterwards. Other games of chance, e.g., gambling contracts, use the same technique.

Wallet. A wallet contract manages digital assets, i.e., cryptocurrencies and tokens, and offers additional features such as shared ownership or daily transaction limits. At August 30, 2019, 3.9 M of 17.9 M (21 %) deployed smart contracts have been different types of wallet contracts [5]. Multi-signature wallets are a special type of wallet that provides a transaction voting mechanism by only executing transactions, which are signed by a fixed fraction of the set of owners. Wallets transfer money and call other contracts in their users stead depending on run-time input, demonstrating calls among contracts in Prisma. Further, a multi-signature wallet uses built-in features of the Ethereum VM for signature validation, i.e., data encoding, hash calculation, and signature verification, showing that these features are supported in Prisma.

Libraries. As the cost of deploying a contract increases with the amount of code in Ethereum, developers try to avoid code repetitions. Contract inheritance does not help: child contracts simply copy the attributes and functions from the parent. Yet, one can outsource commonly used logic to *library contracts* that are deployed once and called by other contracts. For example, the TicTacToe dApp and the TicTacToe channel in our case studies share some logic, e.g., to check the win condition. To demonstrate libraries in Prisma, we include a TicTacToe library to our case studies and another on-chain executed TicTacToe dApp which uses such library instead of deploying the logic itself. Libraries use a call instruction similar to wallets, although the call target is typically known at deployment and can be hard-coded.

Fig. 22. Categories and Cross-tier calls.

Category	Case study	Cross-tier calls	Prisma LoC	Solidity + JavaScript LoC
Financial	Token	4	79	48 + 50
	Crowdfunding	11	59	27 + 63
	Escrow	9	63	33 + 56
Wallet	Multi-signature wallet	3	76	41 + 52
Notary	General-purpose notary	3	32	16 + 36
Game	Rock Paper Scissors	12	79	41 + 77
	TicTacToe	5	61	31 + 52
	Hangman	15	119	86 + 83
	Chinese Checkers	4	167	141 + 47
Library	TicTacToe library	–	167	141 + –
	TicTacToe using library	5	53	29 + 52
Scalability	TicTacToe channel	9	177	56 + 177

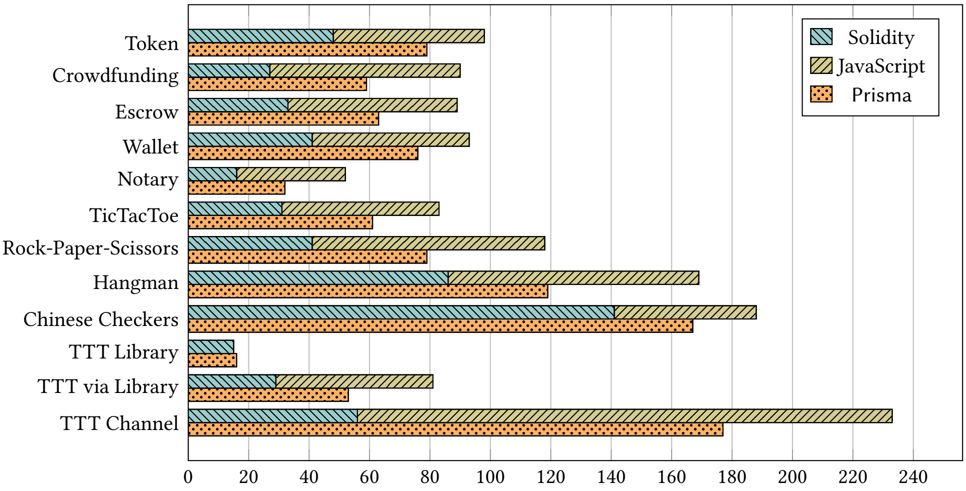


Fig. 23. LOC in Solidity/JavaScript and Prisma.

Scalability solutions. State channels [29, 30, 57] are scalability solutions, which enable a fixed group of parties to move their dApp to a non-blockchain consensus protocol: the execution falls-back to the blockchain in case of disputes. Similar to multi-signature wallets, state channels use built-in signature validation. We implemented a state channel for TicTacToe[12] to demonstrate that Prisma supports state channels.

B EMPIRICAL EVALUATION OF DESIGN QUALITY

In Section 6, we argued that with Prisma, (a) we provide communication safety with a standard system-F-like type-system, (b) the program flow can be defined explicitly and is enforced automatically, (c) dApp developers need to master a single technology that covers both tiers, (d) cross-tier type-safety can be checked at compile-time, and (e) the code is simpler and less verbose due to reduced boilerplate code for communication and less control flow jumps. The claims (a), (c), and (d) are a direct consequence of Prisma's design and do not require further evidence. Claim (c) has been formally proven in Section 3. It remains to investigate claim (e), i.e., in which extent Prisma reduces the amount of code and error-prone control-flow jumps.

To this end, we implemented all case studies with equivalent functionality in Prisma and in Solidity/JavaScript. The JavaScript client logic is in direct style using async/await – the Solidity contract needs to be implemented as a finite-state-machine. We keep the client logic of our case studies (in both, the Prisma and the Solidity implementation) as basic as possible, not to compare the client logic in Scala and in JavaScript but rather focus on the dApp semantics. A complex client logic would shadow the interaction with the contract logic – limited in size due to the gas semantics.

We start with comparing LOCs in the case studies (Figure 23). The results in Figure 23 show that case studies written in Prisma require only 55 – 89 % LOC compared to those implemented in Solidity/JavaScript. One exception is the standalone library, which has no client code and hence does not directly profit from the tierless design.

Second, we consider occurrences of explicit cross-tier control-flow calls in the Solidity/JavaScript dApps (cf. Table 22), which complicate control flow, compared to Prisma, where cross-tier access is seamless. In the client implementations, 6 – 18 % of all lines trigger a contract interaction passing the control flow to the contract and waiting for the control flow to return. From the contract code in finite-state-machine style, it is not directly apparent at which position the program flow continues, once passed back from clients to contract, i.e., which function is called by the clients next. Direct-style code, on the other hand, ensures that the control flow of the contract always continues in the line that passed the control flow to the client by invoking an `awaitCl` expression.

[12] A general solution is a much larger engineering effort and subject of industrial projects [42, 83]

$$
\begin{array}{lcl}
comp'(d; b; \mathsf{trmp}(m)) & = & d;\ coclfn(b, id, \mathsf{assert}(\mathsf{false}), \mathsf{assert}(\mathsf{false}));\ \mathsf{trmp}(comp(m)) \\
 & & \ \text{where} \quad id\ \text{fresh}
\end{array}
$$

$$
\begin{array}{lcl}
comp\left(\begin{array}{l} d; coclfn(b, id, \\ \ e_{1,alt}, \\ \ e_{2,alt}); \\ tmp \leftarrow_s (() \rightarrow e_1);\ e_2 \end{array}\right) & = & \left(\begin{array}{l} d; coclfn(b, id, \\ \ \text{if let } (c :: fv(() \rightarrow e_1)) = id \text{ then } e_1 \text{ else } e'_{1,alt}, \\ \ \text{if let } (c :: x :: fv(x \rightarrow e'_2)) = id \text{ then} \\ \ \mathsf{assert}(\mathsf{this.state} == c\ \&\&\ \mathsf{this.who}(\mathsf{this.sender})); \\ \ \mathsf{this.state} := 0;\ e'_2 \\ \ \text{else} \\ \ e'_{2,alt}); \\ \mathsf{this.who} := e_0;\ \mathsf{this.state} := c; \\ (\mathsf{More},\ c :: fv(() \rightarrow e_1),\ c :: fv(x \rightarrow e'_2)) \end{array}\right) \\
 & & \ \text{where} \quad c\ \text{fresh} \\
 & & \ \text{and} \qquad d; coclfn(b, id, e'_{1,alt}, e'_{2,alt}); e'_2 = \\
 & & \qquad\qquad defun(d; coclfn(b, id, e_{1,alt}, e_{2,alt}); e_2) \\
comp\left(\begin{array}{l} d; coclfn(b, id, e_{1,alt}, e_{2,alt}); \\ x = e_0;\ e_1 \end{array}\right) & = & \left(\begin{array}{l} d; coclfn(b, id, e_{1,alt}, e_{2,alt}); \\ x = e_0;\ defun(e_1) \end{array}\right) \\
comp\left(\begin{array}{l} d; coclfn(b, id, e_{1,alt}, e_{2,alt}); \\ e \end{array}\right) & = & \left(\begin{array}{l} d; coclfn(b, id, e_{1,alt}, e_{2,alt}); \\ e \end{array}\right)
\end{array}
$$

$$
coclfn(b, id, e_{1,alt}, e_{2,alt}) = (@\mathsf{cl}\ \mathsf{this.clfn} = id \rightarrow e_{1,alt}); (@\mathsf{co}\ \mathsf{this.cofn} = id \rightarrow e_{2,alt}); b
$$

Fig. 24. comp′ and comp.

$$
\begin{array}{lcl}
fv(m_0 :: m_1) & = & fv(m_0) \cup fv(m_1) \\
fv(x \rightarrow m) & = & fv(m) \setminus fv(x) \\
fv(id) & = & \{id\} \\
fv(m_0\ m_1) & = & fv(m_0) \cup fv(m_1) \\
fv(\mathsf{awaitCl}^*((m_0, () \rightarrow m_1) & = & fv(m_0) \cup fv(m_1) \\
fv(\mathsf{let}\ x = m_0; m_1) & = & fv(m_0) \cup fv(m_1) \setminus fv(x) \\
fv(\mathsf{this}.i := m_0) & = & fv(m_0) \\
fv(\mathsf{this}.j := m_0) & = & fv(m_0) \\
fv(\mathsf{this}.i) & = & \{\} \\
fv(\mathsf{this}.j) & = & \{\} \\
fv(c) & = & \{\}
\end{array}
$$

Fig. 25. Free variables.

C PROOFS

We provide the definition of comp′ and comp in Figure 24, the definition for the free variables for a given term *fv* in Figure 25 and the detailed proofs for the theorem and the lemmas on the following pages.

Theorem 1 (Secure Compilation). For all programs P over closed terms, the trace set of evaluating the program under attack equals the trace set of evaluating the compiled program under attack, i.e.,

$$\forall P.\ \{\ init_A(comp'(mnf'((P))))\ \} \approx \ldots \approx \{\ init_A(P)\ \}$$

Proof.

$$\begin{aligned} & init_A(comp'(mnf'(P))) \\ \overset{\text{Lemma 3}}{\approx}\ & init_A(mnf'(P)) \\ \overset{\text{Lemma 5}}{\approx}\ & init_A(P) \end{aligned}$$

□

Extensions. For simplicity, our definition of initialization uses a fixed set of clients. Yet, the malicious semantics does not actually depend on the fixed set of clients, but instead models an attacker that is in control of all clients with the capability of sending messages from any client, not bound to the fixed set. Hence, it is straightforward to extend the proofs to the setting of a dynamic set of clients, e.g., clients joining and leaving at run time.

Further, our trace equality relation defines that all programs in the relation eventually reduce to values, filtering out programs that loop or get stuck. Below, we outline an approach to prove trace equality for looping or stuck programs by showing that such programs loop with the same infinite trace or get stuck at the same trace, respectively. To this end, we track the number of steps done via a step-indexed trace equality relation:

$$p; q; e \Downarrow^n =\ \{\ (p', v) \mid (p; q; e) \longrightarrow^n (p'; q'; v)\ \}\ T \Downarrow^n = \bigcup_{p;q;e \in T} p; q; e \Downarrow^n$$

With this definition, we can no longer use just equality of traces as the left and right program may take a different number of steps to produce the same events. Instead, we move from an equality relation to a relation stating non-disagreement, which says that – independently of how long we run either statement – the traces will never be in disagreement:

$$(T \approx^n S)\ \Leftrightarrow\ (T \Downarrow^n\ \#_{\text{set}}\ S \Downarrow^n)$$

where $\#_{\text{set}}$ is defined on trace sets as

$$T \#_{\text{set}} S\ \Leftrightarrow\ (\forall t{\in}T.\ \exists s{\in}S.\ t\ \#_{\text{trace}}\ s) \land (\forall s{\in}S.\ \exists t{\in}T.\ t\ \#_{\text{trace}}\ s)$$

and $\#_{\text{trace}}$ on event traces as

$$\begin{array}{lllllll} (ev, & ()) & \#_{\text{trace}} & (ev, & tail_2) & = true \\ (ev, & tail_1) & \#_{\text{trace}} & (ev, & ()) & = true \\ (ev_1, & tail_1) & \#_{\text{trace}} & (ev_2, & tail_2) & = false \\ (ev, & tail_1) & \#_{\text{trace}} & (ev, & tail_2) & = tail_1\ \#_{\text{trace}}\ tail_2 \end{array} .$$

Lemma 1 (assoc preserves traces). *assoc* is defined as a recursive term-to-term transformation on open terms, whereas traceset equality is defined by reducing terms to values, i.e., on closed terms. Since all valid programs are closed terms, we show that *assoc* preserves the traceset of an open term e that is closed by substitution $[x \Mapsto v]$.

For all terms e, traces p, traces q, values v, patterns x,

$$\{\ p;q;\ \ [x{\Mapsto}v]\ assoc(e)\ \}\ \approx\ \ldots\ \approx\ \{\ p;q;\ \ [x{\Mapsto}v]\ e\ \}$$

Proof. By induction over term structure.

Case. $e\ =\ (\text{let } x_1 = (\text{let } x_0 = e_0;\ e_1);\ e_2)$.

We know $x_0 \notin fv(e_2)$ since e_2 is not in the scope of the x_0 binding, and that all identifiers are distinct, which can always be achieved by α-renaming.

$$x_0 \notin fv(e_2)$$

According to $\approx$, we only consider terms that reduce to a value. Therefore, let ϕ be the judgement that the term e_0 closed by $[x{\Mapsto}v]$ with trace p evaluates to a value v_0 producing trace p_0.

$$\phi\ \equiv\ (p;q;\ [x{\Mapsto}v]e_0 \longrightarrow^{*} p\ p_0;q;\ v_0)$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom ($\longrightarrow^{*}$) and the original program from bottom to top ($\longleftarrow^{*}$) until configurations converge. The induction hypothesis (IH) allows the removal of *assoc* in redex position under traceset equality ($\approx$).

$$\begin{aligned}
& \{\ p;q;\ [x{\Mapsto}v]\ assoc(e) && \} \\
\overset{\text{def. } e}{=}\ & \{\ p;q;\ [x{\Mapsto}v]\ assoc(\text{let } x_1 = (\text{let } x_0 = e_0;\ e_1);\ e_2) && \} \\
\overset{\text{def. } assoc}{=}\ & \{\ p;q;\ [x{\Mapsto}v]\ assoc(\text{let } x_0 = e_0;\ assoc(\text{let } x_1 = e_1;\ e_2)) && \} \\
\overset{IH}{\approx}\ & \{\ p;q;\ [x{\Mapsto}v]\ \text{let } x_0 = e_0;\ assoc(\text{let } x_1 = e_1;\ e_2) && \} \\
\overset{\text{def. } \Mapsto}{=}\ & \{\ p;q;\ \text{let } x_0 = [x{\Mapsto}v]\ e_0;\ [x{\Mapsto}v]\ assoc(\text{let } x_1 = e_1;\ e_2) && \} \\
\overset{\phi}{\longrightarrow^{*}}\ & \{\ p\ p_0;q;\ \text{let } x_0 = v_0;\ [x{\Mapsto}v]\ assoc(\text{let } x_1 = e_1;\ e_2) \mid \forall\ v_0\ p_0,\ \phi && \} \\
\overset{\text{RLet}}{\longrightarrow}\ & \{\ p\ p_0;q;\ [x_0{\Mapsto}v_0,\ x{\Mapsto}v]\ assoc(\text{let } x_1 = e_1;\ e_2) \mid \forall\ v_0\ p_0,\ \phi && \} \\
\overset{IH}{\approx}\ & \{\ p\ p_0;q;\ [x_0{\Mapsto}v_0,\ x{\Mapsto}v]\ \text{let } x_1 = e_1;\ e_2 \mid \forall\ v_0\ p_0,\ \phi && \} \\
\overset{\text{def. } \Mapsto;\ x_0 \notin fv(e_2)}{=}\ & \{\ p\ p_0;q;\ \text{let } x_1 = [x_0{\Mapsto}v_0,\ x{\Mapsto}v]e_1;\ [x{\Mapsto}v]e_2 \mid \forall\ v_0\ p_0,\ \phi && \} \\
\overset{\text{RLet}}{\longleftarrow}\ & \{\ p\ p_0;q;\ \text{let } x_1 = (\text{let } x_0 = v_0;\ [x{\Mapsto}v]e_1);\ [x{\Mapsto}v]e_2 \mid \forall\ v_0\ p_0,\ \phi && \} \\
\overset{\phi}{\longleftarrow^{*}}\ & \{\ p;q;\ \text{let } x_1 = (\text{let } x_0 = [x{\Mapsto}v]e_0;\ [x{\Mapsto}v]e_1);\ [x{\Mapsto}v]e_2 && \} \\
\overset{\text{def. } \Mapsto}{=}\ & \{\ p;q;\ [x{\Mapsto}v]\ \text{let } x_1 = (\text{let } x_0 = e_0;\ e_1);\ e_2 && \} \\
\overset{\text{def. } e}{=}\ & \{\ p;q;\ [x{\Mapsto}v]\ e && \}
\end{aligned}$$

Case. $e\ \neq\ (\text{let } x_1 = (\text{let } x_0 = e_0;\ e_1);\ e_2)$.

If e is not of nested let form, we simply apply the definition of *assoc*.

$$\begin{aligned} & \left\{\; p; q;\; [x \Mapsto v]\; assoc(e) \;\right\} \\ \overset{\text{def. } assoc}{=} & \left\{\; p; q;\; [x \Mapsto v]\; e \;\right\} \end{aligned}$$

$\square$

Lemma 2 (mnf preserves traces). *mnf* is defined as a recursive term-to-term transformation on open terms, whereas traceset equality is defined by reducing terms to values, i.e., on closed terms. Since all valid programs are closed terms, we show that *mnf* preserves the traceset of an open term e that is closed by substitution $[x \Mapsto v]$.

For all terms e, traces p, traces q, values v, patterns x,

$$\{\ p; q;\ [x{\Mapsto}v]\ mnf(e)\ \} \ \approx\ \dots\ \approx\ \{\ p; q;\ [x{\Mapsto}v]\ e\ \}$$

Proof. By induction over term structure.

Case. $e\ =\ e_0\ e_1$.

According to $\approx$, we only consider terms that reduce to a value. Therefore, let ϕ_0 be the judgement that the term e_0 closed by $[x{\Mapsto}v]$ with trace p evaluates to a value v_0 producing trace p_0. Let ϕ_1 be the judgement that the term e_1 closed by $[x{\Mapsto}v]$ with trace $p\ p_0$ evaluates to a value v_1 producing trace $p\ p_0\ p_1$.

$$\begin{aligned}\phi_0\ &\equiv\ (p; q;\ [x{\Mapsto}v]\ e_0 \longrightarrow^* p\ p_0; q;\ v_0)\\ \phi_1\ &\equiv\ (p\ p_0; q;\ [x{\Mapsto}v]\ e_1 \longrightarrow^* p\ p_0\ p_1; q;\ v_1)\end{aligned}$$

Let id_0 be the fresh identifier *mnf* produces.

$$id_0\ \text{fresh}$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom ($\longrightarrow^*$) and the original program from bottom to top ($\longleftarrow^*$) until configurations converge. The induction hypothesis (IH) allows the removal of *mnf* in redex position under traceset equality ($\approx$).

$$\begin{aligned}
& \{\ p; q;\ [x{\Mapsto}v]\ mnf(e) && \}\\
\overset{\text{def. } e}{=}\ & \{\ p; q;\ [x{\Mapsto}v]\ mnf(e_0\ e_1) && \}\\
\overset{\text{def. } mnf}{=}\ & \{\ p; q;\ [x{\Mapsto}v]\ assoc(\text{let } id_0 = mnf(e_0);\ assoc(\text{let } id_1 = mnf(e_1);\ id_0\ id_1)) && \}\\
\overset{\text{Lemma 1}}{\approx}\ & \{\ p; q;\ [x{\Mapsto}v]\ \text{let } id_0 = mnf(e_0);\ assoc(\text{let } id_1 = mnf(e_1);\ id_0\ id_1) && \}\\
\overset{\text{def. } \Mapsto}{=}\ & \{\ p; q;\ \text{let } id_0 = [x{\Mapsto}v]\ mnf(e_0);\ [x{\Mapsto}v]\ assoc(\text{let } id_1 = mnf(e_1);\ id_0\ id_1) && \}\\
\overset{IH}{\approx}\ & \{\ p; q;\ \text{let } id_0 = [x{\Mapsto}v]\ e_0;\ [x{\Mapsto}v]\ assoc(\text{let } id_1 = mnf(e_1);\ id_0\ id_1)) && \}\\
\overset{\phi_0}{\longrightarrow^*}\ & \{\ p\ p_0; q;\ \text{let } id_0 = v_0;\ [x{\Mapsto}v]\ assoc(\text{let } id_1 = mnf(e_1);\ id_0\ id_1) \mid \forall\ v_0\ p_0,\ \text{if } \phi_0 && \}\\
\overset{\text{RLet}}{\longrightarrow}\ & \{\ p\ p_0; q;\ [id_0{\mapsto}v_0,\ x{\Mapsto}v]\ assoc(\text{let } id_1 = mnf(e_1);\ id_0\ id_1) \mid \forall\ v_0\ p_0,\ \text{if } \phi_0 && \}\\
\overset{\text{Lemma 1}}{\approx}\ & \{\ p\ p_0; q;\ [id_0{\mapsto}v_0,\ x{\Mapsto}v]\ \text{let } id_1 = mnf(e_1);\ id_0\ id_1 \mid \forall\ v_0\ p_0,\ \text{if } \phi_0 && \}\\
\overset{\text{def. } \Mapsto}{=}\ & \{\ p\ p_0; q;\ \text{let } id_1 = [id_0{\mapsto}v_0,\ x{\Mapsto}v]\ mnf(e_1);\ v_0\ id_1 \mid \forall\ v_0\ p_0,\ \text{if } \phi_0 && \}\\
\overset{IH}{=}\ & \{\ p\ p_0; q;\ \text{let } id_1 = [id_0{\mapsto}v_0,\ x{\Mapsto}v]\ e_1;\ v_0\ id_1 \mid \forall\ v_0\ p_0,\ \text{if } \phi_0 && \}\\
\overset{id_0\ \text{fresh}}{=}\ & \{\ p\ p_0; q;\ \text{let } id_1 = [x{\Mapsto}v]\ e_1;\ v_0\ id_1 \mid \forall\ v_0\ p_0,\ \text{if } \phi_0 && \}\\
\overset{\phi_1}{\longrightarrow^*}\ & \{\ p\ p_0\ p_1; q;\ \text{let } id_1 = v_1;\ v_0\ id_1 \mid \forall\ v_0\ v_1\ p_0\ p_1,\ \text{if } \phi_0,\ \phi_1 && \}\\
\overset{\text{RLet}}{\longrightarrow}\ & \{\ p\ p_0\ p_1; q;\ v_0\ v_1 \mid \forall\ v_0\ v_1\ p_0\ p_1,\ \text{if } \phi_0,\ \phi_1 && \}
\end{aligned}$$

$$\begin{array}{rl}
\overset{\phi_1}{\longleftarrow^{*}} & \{\; p\ p_0; q;\ v_0\ [x\Mapsto v]\ e_1 \mid \forall\, v_0\ p_0,\ \text{if}\ \phi_0 \;\} \\
\overset{\phi_0}{\longleftarrow^{*}} & \{\; p; q;\ ([x\Mapsto v]\ e_0)\ [x\Mapsto v]\ e_1 \;\} \\
\overset{\text{def.}\,\Mapsto}{=} & \{\; p; q;\ [x\Mapsto v]\ e_0\ e_1 \;\} \\
\overset{\text{def.}\,e}{=} & \{\; p; q;\ [x\Mapsto v]\ e \;\}
\end{array}$$

Case. $e\ =\ \text{let}\ id_0 = e_0;\ e_1$.

According to $\approx$, we only consider terms that reduce to a value. Therefore, let ϕ be the judgement that the term e_0 closed by $[x\Mapsto v]$ with trace p evaluates to a value v_0 producing trace p_0.

$$\phi_0\ \equiv\ (p; q;\ [x\Mapsto v]\ e_0 \longrightarrow^{*} p\ p_0; q;\ v_0)$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom ($\longrightarrow^{*}$) and the original program from bottom to top ($\longleftarrow^{*}$) until configurations converge. The induction hypothesis (IH) allows the removal of mnf in redex position under traceset equality ($\approx$).

$$\begin{array}{rl}
 & \{\; p; q;\ [x\Mapsto v]\ mnf(e) \;\} \\
\overset{\text{def.}\,e}{=} & \{\; p; q;\ [x\Mapsto v]\ mnf(\text{let}\ id_0 = v_0;\ e_1) \;\} \\
\overset{\text{def.}\,mnf}{=} & \{\; p; q;\ [x\Mapsto v]\ assoc(\text{let}\ id_0 = mnf(e_0);\ mnf(e_1)) \;\} \\
\overset{\text{Lemma 1}}{\approx} & \{\; p; q;\ [x\Mapsto v]\ \text{let}\ id_0 = mnf(e_0);\ mnf(e_1) \;\} \\
\overset{\text{def.}\,\Mapsto}{=} & \{\; p; q;\ \text{let}\ id_0 = [x\Mapsto v]\ mnf(e_0);\ [x\Mapsto v]\ mnf(e_1) \;\} \\
\overset{IH}{\approx} & \{\; p; q;\ \text{let}\ id_0 = [x\Mapsto v]\ e_0;\ [x\Mapsto v]\ mnf(e_1) \;\} \\
\overset{\phi_0}{\longrightarrow^{*}} & \{\; p\ p_0; q;\ \text{let}\ id_0 = v_0;\ [x\Mapsto v]\ mnf(e_1) \mid \forall\, v_0\ p_0,\ \text{if}\ \phi_0 \;\} \\
\overset{\textsc{RLet}}{\longrightarrow} & \{\; p\ p_0; q;\ [id_0\mapsto v_0,\ x\Mapsto v]\ mnf(e_1) \mid \forall\, v_0\ p_0,\ \text{if}\ \phi_0 \;\} \\
\overset{IH}{\approx} & \{\; p\ p_0; q;\ [id_0\mapsto v_0,\ x\Mapsto v]\ e_1 \mid \forall\, v_0\ p_0,\ \text{if}\ \phi_0 \;\} \\
\overset{\textsc{RLet}}{\longleftarrow} & \{\; p\ p_0; q;\ [x\Mapsto v]\ \text{let}\ id_0 = v_0;\ e_1 \mid \forall\, v_0\ p_0,\ \text{if}\ \phi_0 \;\} \\
\overset{\phi_0}{\longleftarrow^{*}} & \{\; p; q;\ [x\Mapsto v]\ \text{let}\ id_0 = e_0;\ e_1 \;\} \\
\overset{\text{def.}\,e}{=} & \{\; p; q;\ [x\Mapsto v]\ e \;\}
\end{array}$$

Case. The other cases of e are proved analogously.

□

Lemma 3 (mnf' preserves trace). *mnf'* is defined on programs. To evaluate a program, it is initialized with a set of clients A. *mnf'* preserves the traceset of (closed) programs P for any set of clients A.

For all P,

$$\{\ init_A(mnf'(P))\ \}\ \approx\ \ldots\ \approx\ \{\ init_A(P)\ \}$$

Proof. By induction over term structure.

Case. $P\ =\ (d; b;\ e_0)$.

Initializing the definitions $d; b$ with A produces the trace p and the state q.

$$init_A(d; b)\ =\ p; q$$

According to $\approx$, we only consider terms that reduce to a value. Therefore, let ϕ be the judgement that the term e_0 closed by $[x \mapsto v]$ in trace p produces a value v_0 and trace p_0.

$$\phi\ \equiv\ (p; q;\ e_0 \longrightarrow^*\ p\ p_0; q;\ v_0)$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom ($\longrightarrow^*$) and the original program from bottom to top ($\longleftarrow^*$) until configurations converge, using Lemma 2.

$$\begin{array}{rll}
 & \{\ init_A(mnf'(P)) & \} \\
\overset{\text{def. } P}{=} & \{\ init_A(mnf'(d; b;\ e_0)) & \} \\
\overset{\text{def. } mnf'}{=} & \{\ init_A(d; b;\ \mathsf{trmp}(mnfe(\mathsf{Done}(e_0)))) & \} \\
\overset{\text{def. } init_A}{=} & \{\ p; q;\ \mathsf{trmp}(mnfe(\mathsf{Done}(e_0))) & \} \\
\overset{\text{Lemma 2}}{\approx} & \{\ p; q;\ \mathsf{trmp}(\mathsf{Done}(e_0)) & \} \\
\overset{\phi}{\longrightarrow^*} & \{\ p\ p_0; q;\ \mathsf{trmp}(\mathsf{Done}(v_0))\ \mid \forall\, v_0\ p_0,\ \mathsf{if}\ \phi & \} \\
\overset{\text{RDone}}{\longrightarrow} & \{\ p\ p_0; q;\ v_0\ \mid \forall\, v_0\ p_0,\ \mathsf{if}\ \phi & \} \\
\overset{\phi}{\longleftarrow^*} & \{\ p; q;\ e_0 & \} \\
\overset{\text{def. } init_A}{=} & \{\ init_A(d; b;\ e_0) & \} \\
\overset{\text{def. } P}{=} & \{\ init_A(P) & \}
\end{array}$$

□

Lemma 4 (comp preserves traces). *comp* is defined on programs. To evaluate a program, it is initialized with a set of clients A. *comp* preserves the traceset of (closed) programs P for any set of clients A.

For all definitions b, definitions d, terms e, values v, patterns x,

$$\{\ [x{\Mapsto}v]\ init_A(comp(d;b;\ \mathsf{trmp}(e)))\ \}\ \approx\ \dots\ \approx\ \{\ init_A(d;b;\ \mathsf{trmp}([x{\Mapsto}v]\ e))\ \}$$

Proof. By induction over term structure.

Case. $e\ =\ \mathsf{let}\ x = \mathsf{awaitCl}_s((e_0,\ ()\ {\longrightarrow} e_1));\ e_2.$
comp expects the definitions b to be of form:

$$b\ =\ \left(\begin{array}{l} \text{@cl this.clfn}\ =\ id \longrightarrow e_{1,alt};\\ \text{@co this.cofn}\ =\ id \longrightarrow e_{2,alt};\\ b_{rest} \end{array}\right)$$

comp is defined recursively and applied to the term e_2. Intuitively, *comp* transforms e_2 to e_2' and b to b' by moving the part of e_2 that comes after the $\mathsf{awaitCl}_s$ call into the cofn definition inside b. The recursive call is given as follows:

$$(d;b';\ \mathsf{trmp}(e_2'))\ =\ comp(d;b;\ \mathsf{trmp}(e_2))$$

$$b'\ =\ \left(\begin{array}{l} \text{@cl this.clfn}\ =\ id \longrightarrow e'_{1,alt};\\ \text{@co this.cofn}\ =\ id \longrightarrow e'_{2,alt};\\ b_{rest} \end{array}\right)$$

After the recursive call, *comp* moves the transformed e_2' into the cofn definition, resulting in e' and b'' with $e''_{1,alt}$ and $e''_{2,alt}$.

$$\phi\ \equiv\ \big(\ \{\ d;b'';\ \mathsf{trmp}(e')\ \big\}\ =\ \big\{\ comp(d;b;\ \mathsf{trmp}(e))\ \}\ \big)$$

$$b''\ =\ \left(\begin{array}{l} \text{@cl this.clfn}\ =\ id \longrightarrow e''_{1,alt};\\ \text{@co this.cofn}\ =\ id \longrightarrow e''_{2,alt};\\ b_{rest} \end{array}\right)$$

$$e''_{1,alt}\ =\ \left(\begin{array}{l} \mathsf{if\ let}\ (c :: fv(()\ {\longrightarrow} e_1))\ =\ id\\ \mathsf{then}\ e_1\\ \mathsf{else}\ e'_{2,alt} \end{array}\right)$$

$$e''_{2,alt}\ =\ \left(\begin{array}{l} \mathsf{if\ let}\ (c :: x :: fv(x\ {\longrightarrow} e_2'))\ =\ id\\ \mathsf{then\ assert(this.state ==}\ c\ \&\&\ \mathsf{this.sender == this.who);}\\ \mathsf{this.state := 0;}\ e_2'\\ \mathsf{else}\ e'_{2,alt} \end{array}\right)$$

Let $p;q$ be the trace and state produced by initializing $d;b$ with A, and $p;q'$ for initializing $d;b'$, and $p;q''$ for initializing $d;b''$.

$$\begin{aligned} init_A(d;b)\ &=\ p;q\\ init_A(d;b')\ &=\ p;q'\\ init_A(d;b'')\ &=\ p;q'' \end{aligned}$$

According to $\approx$, we only consider terms that reduce to a value. Therefore, let ϕ_0 be the judgement that the term e_0 closed by $[x{\Mapsto}v]$ in trace p produces a value v_0 and trace p_1.

$$\phi_0(q_\phi)\ =\ (p;q_\phi;\ [x{\Mapsto}v]\ e_0 \longrightarrow p\ p_1;q_\phi;\ v_0)$$

We define ϕ_1 based on ϕ:

$$
\begin{array}{c}
\phi \\
= \\
\{\, d; b''; \ \mathsf{trmp}(e') \,\} \ = \ \{\, \mathit{comp}(d; b; \ \mathsf{trmp}(e)) \,\} \\
\longrightarrow \mathit{generalize} \ [x \Mapsto v] \ \mathit{init}_A(...) \\
\{\, [x \Mapsto v] \ \mathit{init}_A(d; b''; \ \mathsf{trmp}(e')) \,\} \ = \ \{\, [x \Mapsto v] \ \mathit{init}_A(\mathit{comp}(d; b; \ \mathsf{trmp}(e))) \,\} \\
\longrightarrow (= \ \longrightarrow \ \approx) \\
\{\, [x \Mapsto v] \ \mathit{init}_A(d; b''; \ \mathsf{trmp}(e')) \,\} \ \approx \ \{\, [x \Mapsto v] \ \mathit{init}_A(\mathit{comp}(d; b; \ \mathsf{trmp}(e))) \,\} \\
\longrightarrow \mathit{IH} \\
\{\, [x \Mapsto v] \ \mathit{init}_A(d; b''; \ \mathsf{trmp}(e')) \,\} \ \approx \ \{\, \mathit{init}_A(d; b; \ \mathsf{trmp}([x \Mapsto v] \ e)) \,\} \\
\longrightarrow \text{def. } \mathit{init}_A \\
\{\, p; q''; \ \mathsf{trmp}([x \Mapsto v] \ e') \,\} \ \approx \ \{\, p; q; \ \mathsf{trmp}([x \Mapsto v] \ e) \,\} \\
\equiv \\
\phi_1
\end{array}
$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom ($\longrightarrow^*$) and the original program from bottom to top ($\longleftarrow^*$) until configurations converge.

$$
\begin{array}{cl}
 & \left\{ \ [x \Mapsto v] \ \mathit{init}_A(\mathit{comp}(d; b; \ \mathsf{trmp}(e))) \right\} \\[1ex]
\overset{\text{def. } e}{=} & \left\{ \ [x \Mapsto v] \ \mathit{init}_A(\mathit{comp}(d; b; \ \mathsf{trmp}(\mathsf{let}\ x_3 = \mathsf{awaitCl}_s((e_0, \ () \rightarrow e_1))); \ e_2)) \right\} \\[1ex]
\overset{\text{def. } \mathit{comp}}{=} & \left\{ \begin{array}{l} [x \Mapsto v] \ \mathit{init}_A(d; b''; \ \mathsf{trmp}(\\ \mathsf{this.who} := e_0; \ \mathsf{this.state} := c; \\ \mathsf{More}(c :: \ \mathit{fv}(() \rightarrow e_1), \ c :: \ \mathit{fv}(x \rightarrow e_2')))) \end{array} \right\} \\[1ex]
\overset{\text{def. } \mathit{init}_A}{=} & \left\{ \begin{array}{l} p; q''; \ [x \Mapsto v] \ \mathsf{trmp}(\\ \mathsf{this.who} := e_0; \ \mathsf{this.state} := c; \\ \mathsf{More}(c :: \ \mathit{fv}(() \rightarrow e_1), \ c :: \ \mathit{fv}(x \rightarrow e_2')) \end{array} \right\} \\[1ex]
\overset{\text{def. } \Mapsto}{=} & \left\{ \begin{array}{l} p; q''; \ \mathsf{trmp}(\\ \mathsf{this.who} := [x \Mapsto v] \ e_0; \ \mathsf{this.state} := c; \\ \mathsf{More}(c :: [x \Mapsto v] \ \mathit{fv}(() \rightarrow e_1), \ c :: [x \Mapsto v] \ \mathit{fv}(x \rightarrow e_2'))) \end{array} \right\} \\[1ex]
\overset{\phi_0(q'')}{\longrightarrow^*} & \left\{ \begin{array}{l} p \ p_1; \ q''; \ \mathsf{trmp}(\\ \mathsf{this.who} := v_0; \ \mathsf{this.state} := c; \\ \mathsf{More}(c :: [x \Mapsto v] \ \mathit{fv}(() \rightarrow e_1), \ c :: [x \Mapsto v] \ \mathit{fv}(x \rightarrow e_2'))) \\ \mid \forall\, v_0 \ p_1, \ \mathsf{if} \ \phi_0 \end{array} \right\} \\[1ex]
\overset{\text{RSET}\dagger, \ \text{RSET}\dagger}{\longrightarrow} & \left\{ \begin{array}{l} p \ p_1; \ q'' \ [\mathsf{who} \mapsto v_0, \ \mathsf{state} \mapsto c]; \\ \mathsf{trmp}(\mathsf{More}(c :: [x \Mapsto v] \ \mathit{fv}(() \rightarrow e_1)), \ c :: [x \Mapsto v] \ \mathit{fv}(x \rightarrow e_2')) \\ \mid \forall\, v_0 \ p_1, \ \mathsf{if} \ \phi_0 \end{array} \right\} \\[1ex]
\overset{\text{RMORE}}{\longrightarrow} & \left\{ \begin{array}{l} p \ p_1; \ q'' \ [\mathsf{who} \mapsto v_0, \ \mathsf{state} \mapsto c]; \\ \mathit{tmp} \leftarrow_t \mathsf{this.clfn}(c :: [x \Mapsto v] \ \mathit{fv}(() \rightarrow e_1)); \\ \mathsf{trmp}(\mathsf{this.cofn}(c :: \mathit{tmp} :: [x \Mapsto v] \ \mathit{fv}(x \rightarrow e_2'))) \\ \mid \forall\, v_0 \ p_1, \ \mathsf{if} \ \phi_0 \end{array} \right\} \\[1ex]
\overset{\text{RBT}}{\longrightarrow} & \left\{ \begin{array}{l} p \ p_1 \ \mathsf{msg}(v_0', v_2') \ \mathsf{wr}(0, \mathsf{sender}, v_0'); \ q'' \ [\mathsf{who} \mapsto v_0, \ \mathsf{state} \mapsto c]; \\ \mathsf{let}\ \mathit{tmp} = v_2'; \ \mathsf{trmp}(\mathsf{this.cofn}(c :: \mathit{tmp} :: [x \Mapsto v] \ \mathit{fv}(x \rightarrow e_2'))) \\ \mid \forall\, v_0 \ p_1 \ v_0' \ v_2', \ \mathsf{if} \ \phi_0 \end{array} \right\}
\end{array}
$$

$$\overset{\mathit{case}\ v_0' = v_0}{=} \left\{ \begin{array}{l} p\ p_1\ \mathsf{msg}(v_0', v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0');\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto c]; \\ \mathsf{let}\ tmp = v_2';\ \mathsf{trmp}(\mathsf{this.cofn}(c :: tmp :: [x \Mapsto v]\ fv(x \rightarrow e_2'))) \\ \mid \forall\, v_0\ p_1\ v_0'\ v_2',\ \mathsf{if}\ v_0' \neq v_0,\ \phi_0 \\ \hline p\ p_1\ \mathsf{msg}(v_0', v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0');\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto c]; \\ \mathsf{let}\ tmp = v_2';\ \mathsf{trmp}(\mathsf{this.cofn}(c :: tmp :: [x \Mapsto v]\ fv(x \rightarrow e_2'))) \\ \mid \forall\, v_0\ p_1\ v_0'\ v_2',\ \mathsf{if}\ v_0' = v_0,\ \phi_0 \end{array} \right\}$$

$$\overset{\substack{\textsc{RLet}, \textsc{RGet}, \textsc{RApp}, \textsc{Rt}, \\ \textsc{RGet}, \textsc{ROp}, \textsc{RGet}, \\ \textsc{RGet}, \textsc{ROp}, \textsc{ROp}}}{\longrightarrow^*} \left\{ \begin{array}{l} p\ p_1\ \mathsf{msg}(v_0', v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0');\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto c]; \\ \mathsf{trmp}(\mathsf{assert}(\mathsf{false});\ \mathsf{this.state} := 0;\ [x \Mapsto v_2',\ x \Mapsto v]\ e_2')) \\ \mid \forall\, v_0\ p_1\ v_0'\ v_2',\ \mathsf{if}\ v_0' \neq v_0,\ \phi_0 \\ \hline p\ p_1\ \mathsf{msg}(v_0', v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0');\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto c]; \\ \mathsf{trmp}(\mathsf{assert}(\mathsf{true});\ \mathsf{this.state} := 0;\ [x \Mapsto v_2',\ x \Mapsto v]\ e_2') \\ \mid \forall\, v_0\ p_1\ v_0'\ v_2',\ \mathsf{if}\ v_0' = v_0,\ \phi_0 \end{array} \right\}$$

$$\overset{\text{def.} \approx}{\approx} \left\{ \begin{array}{l} p\ p_1\ \mathsf{msg}(v_0', v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0');\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto c]; \\ \mathsf{trmp}(\mathsf{assert}(\mathsf{true});\ \mathsf{this.state} := 0;\ [x \Mapsto v_2',\ x \Mapsto v]\ e_2') \\ \mid \forall\, v_0\ p_1\ v_0'\ v_2',\ \mathsf{if}\ v_0' = v_0,\ \phi_0 \end{array} \right\}$$

$$\overset{\textsc{RLet}, \textsc{RSet}}{\longrightarrow} \left\{ \begin{array}{l} p\ p_1\ \mathsf{msg}(v_0', v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0');\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto 0]; \\ \mathsf{trmp}([x \Mapsto v_2',\ x \Mapsto v]\ e_2') \\ \mid \forall\, v_0\ p_1\ v_0'\ v_2',\ \mathsf{if}\ v_0' = v_0,\ \phi_0 \end{array} \right\}$$

$$\overset{v_0' = v_0}{=} \left\{ \begin{array}{l} p\ p_1\ \mathsf{msg}(v_0, v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0);\ q''\ [\mathsf{who} \mapsto v_0,\ \mathsf{state} \mapsto 0]; \\ \mathsf{trmp}([x \Mapsto v_2',\ x \Mapsto v]\ e_2') \\ \mid \forall\, v_0\ p_1\ v_2',\ \mathsf{if}\ \phi_0 \end{array} \right\}$$

$$\overset{\phi_1}{\approx} \left\{ \begin{array}{l} p\ p_1\ \mathsf{msg}(v_0, v_2')\ \mathsf{wr}(0, \mathsf{sender}, v_0);\ q; \\ \mathsf{trmp}([x \Mapsto v_2',\ x \Mapsto v]\ e_2) \\ \mid \forall\, v_0\ p_1\ v_2',\ \mathsf{if}\ \phi_0 \end{array} \right\}$$

$$\overset{\textsc{RLet}, \textsc{Rbs}}{\longleftarrow} \left\{ \begin{array}{l} p\ p_1;\ q;\ \mathsf{trmp}(\mathsf{let}\ x = \mathsf{awaitCl}_s(v_0,\ () \rightarrow [x \Mapsto v]\ e_1);\ [x \Mapsto v]\ e_2) \\ \mid \forall\, v_0\ p_1,\ \mathsf{if}\ \phi_0 \end{array} \right\}$$

$$\overset{\phi_0(q)}{\longleftarrow^*} \left\{\ p;q;\ \mathsf{trmp}(\mathsf{let}\ x = \mathsf{awaitCl}_s([x \Mapsto v]\ e_0,\ () \rightarrow [x \Mapsto v]\ e_1);\ [x \Mapsto v]\ e_2)\ \right\}$$

$$\overset{\text{def.} \Mapsto}{=} \left\{\ p;q;\ \mathsf{trmp}([x \Mapsto v]\ \mathsf{let}\ x = \mathsf{awaitCl}_s(e_0,\ () \rightarrow e_1);\ e_2)\ \right\}$$

$$\overset{\text{def.}\ e}{=} \left\{\ p;q;\ \mathsf{trmp}([x \Mapsto v]\ e)\ \right\}$$

$$\overset{\text{def.}\ \mathit{init}_A}{=} \left\{\ \mathit{init}_A(b;d;\ \mathsf{trmp}([x \Mapsto v]\ e))\ \right\}$$

Case. $e = x_0$.
Let $p;q$ be the trace and state produced by initializing $d;b$ with A.

$$\mathit{init}_A(d;b) = p;q$$

The traceset equality holds by definition of *comp* and init_A.

$$\left\{\ [x \Mapsto v]\ \mathit{init}_A(\mathit{comp}(d;b;\ \mathsf{trmp}(e)))\ \right\}$$

$$\overset{\text{def.}\ e}{=} \left\{\ [x \Mapsto v]\ \mathit{init}_A(\mathit{comp}(d;b;\ \mathsf{trmp}(x_0)))\ \right\}$$

$$\overset{\text{def.}\ \mathit{comp}}{=} \left\{\ [x \Mapsto v]\ \mathit{init}_A(d;b;\ \mathit{comp}(\mathsf{trmp}(x_0)))\ \right\}$$

$$\overset{\text{def.}\ \mathit{init}_A}{=} \left\{\ p;q;\ [x \Mapsto v]\ \mathsf{trmp}(x_0)\ \right\}$$

$$\begin{aligned}
&\overset{\text{def.}}{=}^{\Mapsto} && \Big\{\ p;q;\ \mathsf{trmp}([x{\Mapsto}v]\ x_0) && \Big\} \\
&\overset{\text{def.}}{=}^{e} && \Big\{\ p;q;\ \mathsf{trmp}([x{\Mapsto}v]\ e) && \Big\} \\
&\overset{\text{def.}}{=}^{init_A} && \Big\{\ init_A(d;b;\ \mathsf{trmp}([x{\Mapsto}v]\ e)) && \Big\}
\end{aligned}$$

□

Lemma 5 (comp' preserves traces). *comp′* is defined on programs. To evaluate a program, it is initialized with a set of clients A. *comp′* preserves the traceset of (closed) programs P for any set of clients A.

For all definitions b, definitions d, terms e_0,

$$\{ \; init_A(comp'(d; b;\; \mathsf{trmp}(e_0))) \; \} \;\approx\; \dots \;\approx\; \{ \; init_A(d; b';\; \mathsf{trmp}(e_0)) \; \}$$

Proof. By induction over term structure.

Case. $P \;=\; (d; b;\; e_0)$.

Intuitively, *comp′* prepends the definitions b with initial definitions for clfn and cofn that only contain assert(false), such that *comp* can be applied.

$$b' \;=\; \left(\begin{array}{l} @\mathsf{cl}\ \mathsf{this.clfn} \;=\; id \longrightarrow \mathsf{assert(false)}; \\ @\mathsf{co}\ \mathsf{this.cofn} \;=\; id \longrightarrow \mathsf{assert(false)}; \\ b \end{array} \right)$$

The lemma holds by definition of *comp′*, and Lemma 4.

$$\begin{array}{cl} & \left\{ \; init_A(comp'(d; b;\; \mathsf{trmp}(e_0))) \; \right\} \\ \overset{\text{def. } comp'}{=} & \left\{ \; init_A(comp(d; b';\; \mathsf{trmp}(e_0)) \; \right\} \\ \overset{\text{Lemma 4}}{\approx} & \left\{ \; init_A(d; b';\; \mathsf{trmp}(e_0)) \; \right\} \end{array}$$

□