Quartz: A Framework for Engineering Secure Smart Contracts

John Kolb
John Yang
Randy H. Katz
David E. Culler

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-178
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-178.html

August 31, 2020
Quartz: A Framework for Engineering Secure Smart Contracts

John Kolb
Computer Science Division
University of California, Berkeley
jkolb@cs.berkeley.edu

Randy H. Katz
Computer Science Division
University of California, Berkeley
randy@cs.berkeley.edu

John B. Yang
Computer Science Division
University of California, Berkeley
john.yang20@berkeley.edu

David E. Culler
Computer Science Division
University of California, Berkeley
culler@cs.berkeley.edu

ABSTRACT
We present Quartz, a language and framework for developing and validating smart contracts. A user writes a Quartz contract as an extended finite-state machine in a domain-specific language. Quartz translates this description into a formal specification in TLA+. The specification captures both the contract’s logic and the execution semantics of the blockchain. Quartz uses bounded model checking to automatically determine if a contract adheres to user-defined properties. Once validated, the contract’s author uses Quartz to generate a Solidity implementation for deployment on a blockchain.

We used a survey of 16 contract case studies, drawn from a variety of domains, to motivate the design of Quartz’s DSL and to evaluate its effectiveness. We show that Quartz contracts are consistently more concise than handwritten Solidity equivalents. We present two in-depth case studies where Quartz identifies multiple significant contract vulnerabilities and provides useful execution traces to assist the developer in addressing them. Finally, we demonstrate that Quartz imposes only modest overhead in terms of generated code size and execution efficiency over handwritten Solidity. Quartz is able to automatically verify functional, contract-specific properties. It is the first tool of this type that detects vulnerabilities arising from interactions with external code, such as reentrancy.

1 INTRODUCTION
Recent developments in blockchains have generated renewed interest in distributed ledger systems. Originally conceived as an immutable and tamper-proof record of financial transactions, blockchains have expanded into the role of secure and robust data repositories for general-purpose applications with the advent of smart contracts. A smart contract defines a body of data and the logic for a set of transformations that may be applied to this data. A contract’s initial state and all subsequent transformations are recorded on a distributed ledger, allowing any consumer of the ledger to inspect the current state of the contract and to validate the history of operations that produced this state. Proper operation of the ledger is ensured by a protocol leveraging cryptographic primitives in which each participant acts out of self interest.

A smart contract operates under the same assumptions — it defines a protocol for multiple distrusting, self-interested parties to execute transactions against a shared body of state. Users rely on the underlying ledger to enforce this protocol by faithfully executing the contract’s logic precisely as written. It is therefore up to the contract’s code to ensure the integrity of its data and to enforce the application-level guarantees its users expect. The correctness and security of the contract’s code is therefore a central concern for its developers and users.

Smart contracts offer powerful capabilities but have proven difficult to engineer for security and correctness. Serious vulnerabilities have been discovered in smart contracts deployed in production systems, even those written by domain experts [44]. The execution semantics for contract code have significant differences from those of more traditional software, making it difficult to identity potential vulnerabilities. A contract’s code is immutable once deployed to a ledger, meaning issues found only after release cannot be patched. Moreover, the internals of any deployed smart contract are open to all of a ledger’s users, who are also free to invoke arbitrary transactions against the contract at any time. Anyone may therefore inspect a contract for vulnerabilities and then attempt to exploit them. There is significant incentive to do so because contracts are often used to manage sensitive data and track valuable assets.

Given these concerns, there is significant value in representing a contract clearly and concisely. Second, a contract’s authors would like to understand and verify as much as possible about its potential behaviors and vulnerabilities before it is deployed to the open environment of a distributed ledger. Third, the authors need the ability to generate a working implementation ready for use from their concise contract description.

This paper presents Quartz, a framework for engineering secure smart contracts that addresses these three goals. The design of this framework is shown in Figure 1. Quartz offers a domain-specific...
language in which a developer describes her contract as an extended finite-state machine. The transitions of this state machine become the transactions against the contract recorded on the underlying ledger. The contract author augments her description with a set of invariants regarding the contract’s behavior. Quartz translates a state machine and invariants into a formal specification suitable for model checking, expressed using TLA+ [34]. A model checker then searches possible contract execution paths for violations of the stated invariants. If a violation is found, the model checker presents the contract author with an execution trace that induces it.

Once the contract obeys the checked invariants, possibly after an iterative process of refining the contract and rerunning the model checker, the author may use Quartz to generate a Solidity implementation from the original state machine description. This allows her to seamlessly deploy the contract to any Ethereum-backed blockchain, with greater assurance that what is deployed in production will behave as expected. Quartz leverages its smart contract description language to make smart contracts both easier to express, facilitating contract development, and easier to analyze, facilitating systematic and robust contract testing. The design of this DSL is informed by a survey of 16 case studies based on real contracts or contract standards spanning a diverse array of domains. These case studies also form the basis of our evaluation. We measure the costs of using Quartz in terms of the amount of code needed to express a contract compared to its Solidity and TLA+ equivalents, measure the execution efficiency of contracts generated by Quartz compared to handwritten alternatives, and present in-depth studies of applying Quartz and its model checking to contract development.

While there have been many efforts to ease contract development and to apply techniques from formal methods to contracts, Quartz is one of the only systems offering fully automated verification of functional properties, i.e., contract-specific invariants, rather than a fixed set of vulnerabilities. One important source of security vulnerabilities is interaction with external code [23]. Quartz is fully capable of modeling these interactions, distinguishing it from similar tools. This allows Quartz to flag issues underlying attacks like the widely-publicized compromise of the DAO contract [13].

The remainder of this paper will motivate, discuss, and evaluate the design of Quartz in more detail. Section 2 covers background material and presents the contract case studies used to motivate the design of Quartz’s language, presented in Section 3. Quartz’s model checking process is described in Section 4, and Section 5 covers Solidity generation. We present our evaluation in Section 6. Finally, Section 7 summarizes related work and Section 8 concludes.

## 2 BACKGROUND & REQUIREMENTS

In this section, we first summarize the execution semantics of contracts on the Ethereum blockchain. We then present a selection of case studies used to inform the design and feature set of the Quartz DSL. We identify common contract design patterns and motivating use cases for specific language primitives.

### 2.1 Ethereum Contract Execution

Contracts in Ethereum are expressed as bytecode for the Ethereum Virtual Machine (EVM), although contracts are mainly written in Solidity, a higher-level language compiled to bytecode. Ethereum’s distributed ledger is maintained by a peer-to-peer network of nodes each running a local instance of the EVM. Each item added to the ledger triggers an execution of bytecode, which enacts some transformation against a smart contract. All participants stay synchronized to the latest state of the blockchain by reaching consensus on the sequence of transactions and then locally executing the corresponding bytecode. Ethereum meters bytecode execution by associating each EVM operation with a “gas cost.” An upper limit on the total gas cost of each transaction (ledger item) ensures termination, and administering a transaction fee based on its total gas discourages loading the network with spurious computations.

Ethereum treats the exchange of virtual tokens, “Ether,” as a first-class primitive. Every contract deployed to the ledger has an associated token balance, and these tokens are used to pay the transaction fees described above. End users may also create accounts that are recorded on the ledger and maintain a balance of tokens on their behalf but are not associated with any executable code. Both these accounts and smart contracts are uniquely identified by a 256-bit public key, known as an address. A contract transformation may involve sending tokens in its possession to an address, which could be either a simple user account or another contract. The latter case is much like a function call, in that control transfers to the receiving contract, which may execute its own code. Thus, the execution of contract code is triggered by an end user explicitly invoking a transaction against that contract or implicitly by the receipt of tokens.

### 2.2 Contract Case Studies

Table 1 summarizes the contracts we studied in detail to motivate and refine Quartz’s language design. Our case studies were drawn from Ethereum standards, subjects of prior work, and real Solidity projects. These contracts encompass a diverse range of use cases such as financial applications (ERC-20, Auction), voting and decision making (ERC-1202, DAO), and data provenance (Logistics). Several involve intricate security concerns and rich feature sets.

We identified several common and recurring design patterns in these contracts, generally in line with prior work [4, 45]:

- **Contracts** typically go through a multi-phase lifecycle. In each phase, a different set of transactions are permitted.
- Authorization is a primary concern for contract code. Most actions are only intended to be performed by a specific set of principals.
- Timing is also a major concern. Some actions are only valid within a specific time range.
- Actions may be conditionally enabled, depending on the contract’s internal state.

In short, contract use cases typically center on maintaining internal state and restricting when, how, and by whom this state is modified, in an effort to adhere to a known protocol.

Additionally, we observed frequent use of certain language features in the implementations of these contracts:

- Both signed and unsigned integer arithmetic, the latter frequently to implement blockchain-managed tokens or assets
- Use of structs to simplify management of complex state and use of maps to track possession of virtual tokens or shares
3 LANGUAGE DESIGN

This section presents the design of the Quartz domain-specific language. We first describe the structure of a Quartz contract description and then present a running example contract that we will use throughout the paper. Next, we formalize Quartz’s syntax and operational semantics.

3.1 Language Structures

A contract definition in Quartz consists of an extended finite-state machine and an optional set of properties to verify about the machine’s behavior. A state machine in turn consists of three pieces. The first is an optional sequence of definitions of any Struct types to be used within the contract. The second piece is a set of fields, each given a unique name and annotated with a type. Quartz supports simple types such as Int and UInt, parameterized types such as Maps and Sequences, and Struct types. Quartz also includes types specifically useful for contract development such as Identity (a unique identifier for a ledger participant) and a TimeSpan type. A HashValue type is parameterized by a sequence of types indicating the structure of its preimage. It only supports equality checks with instances of the same type. This encourages the use of hashing for purposes like commitment schemes and capability-based access control while discouraging the use of hashing as a pseudo-random number generator, a practice that has introduced vulnerabilities in past contracts [3].

The last component of a state machine is a set of state transitions. Each transition is comprised of the following elements:

- A unique name, used to invoke the transition
- A source state and destination state
- A set of parameters, each given a name and type
- A guard, written as a predicate over the machine’s fields and the transition’s parameters
- An authorization predicate restricts which parties may trigger a transition
- A body, written as a sequence of statements executed for their side effects

Quartz state machines are event triggered, and a transition is only eligible for execution if its guard is satisfied. The statements within a transition body are kept simple to facilitate model checking, with no branching constructs. They may either modify a field or transfer tokens to an external contract. An authorization predicate determines who may initiate execution of the associated transition, a particularly important concern for smart contract applications. Quartz’s authorization clauses allow contract authors to express rich semantics that are cumbersome to express using guards alone. They are built from three terms of the form $i$, satisfied when Identity $i$ approves the transition, and the forms any$(I)$ and all$(I)$, where $I$ is of type Sequence[Identity]. These are satisfied when one or all members of the referenced group approve, respectively. These terms may be arbitrarily combined with Boolean && and || operators.

The second, optional element of a contract description is a set of invariants regarding the state machine’s possible execution traces. These are written as predicates over the state machine’s fields, with some additional primitives. Predicates may refer to transition parameter values or to an aggregate sum over a Sequence or Mapping type. Additionally, a predicate can use min or max to refer to the minimum or maximum value that a variable assumes over the lifetime of the state machine. For example, given a state machine containing transition $t$ with parameter $p$, max$(t.p)$ refers to the maximum value of $p$ ever used in an execution of $t$. This allows Quartz to check rudimentary temporal properties [41, 49].

3.2 Running Example

We will use the running example of a smart contract for administering a simple auction on a blockchain. The purpose of this contract is to accept a sequence of ascending bids, each backed by a token deposit, from any potential party for an item put up by a specific seller, who is responsible for deploying the contract. It is up to the contract’s implementation code, which is executed exactly as written by the underlying distributed ledger, to properly enforce the auction’s terms. First, the issuer of the highest bid, regardless of their identity, is duly recorded as the winner. Bids are accepted up
until a publicly declared deadline, which cannot be adjusted after the start of the auction by any party. Any principal who issued a losing bid is able to recover their tokens, while the seller may only claim the auction’s proceeds once it is closed.

Our auction features four phases of operation. Its representation as a state machine is shown in Figure 2. Each transition is annotated with a guard, written above the horizontal line, and actions shown below the horizontal line. When the contract is deployed, its creator is recorded as the seller. The contract begins its life in the Init phase, awaiting the first bid. As soon as a bid arrives, the bid and its sender are recorded, and the contract transitions to the Open phase. Here, an arbitrary number of subsequent bids may be received and recorded, as long as each exceeds the previous highest bid. Unlike in the previous phase, the contract now must also refund the newly-supplanted highest bidder. Finally, once the auction’s deadline has passed, any party may move to close the auction. Only once the Closed phase is reached can the seller claim their earnings, with a transition into the Redeemed phase.

### 3.3 Language Syntax

Figure 3 gives the Quartz implementation of the auction state machine shown above. A formal presentation of Quartz’s syntax is also given in Appendix A. Quartz supports the standard arithmetic and Boolean operators, comparisons, and both in and not in operators to check for membership in an object of Sequence type. Literals of type Bool, Int, Uint, and Timespan are written as expected, with the possible exception of a Timespan instance, written as an integer followed by a unit such as minutes or hours.

Quartz transitions begin with a header of the form source -> (Parameters) destination. This is followed by an optional requires block to express a guard and an optional authorized block to express an authorization predicate. Finally, the body of the transition is enclosed within braces and consists of a sequence of simple statements, like assignment to a field.

The Quartz language contains several contract-specific features. Many distributed ledgers, most notably Ethereum, have first-class support for virtual currency that may be bound to contracts and exchanged among them. State machines in Quartz use keywords to check their balance or disburse tokens to an external contract. If we wish to produce contracts for a ledger without first-class tokens, we can emulate this functionality by adding an extra field and the necessary operations to the generated implementations. Transition authorization is treated as a first-class primitive in Quartz, unlike in Solidity and other contract languages. Quartz allows contract authors to express rich authorization constraints such as restricting an operation to any member of a particular group or requiring approval from all members of a group before it is executed. Finally, Quartz restricts communication between state machines. A state machine may send tokens to another state machine, but it cannot invoke another machine’s transitions directly. This simplifies the expression and verification of contract logic. Note that Quartz makes no assumptions about the behavior of the recipient, which may or may not be another Quartz state machine, for model checking.

```plaintext
contract Auction {
  data {
    Seller: Identity
    HighestBid: Uint
    HighestBidder: Identity
    Deadline: Timestamp
    Duration: Timespan
  }

  initialize: ->(duration: Timespan) init {
    Seller = sender
    Duration = duration
    HighestBid = 0
  }

  submitBid: open ->(tokens: Uint) open {
    HighestBid = tokens
    HighestBidder = sender
  }

  close: open -> closed requires [ now > Deadline ]
    send HighestBid to HighestBidder
  }

  redeem: closed -> redeemed authorized [ Seller ]
    send HighestBid to Seller
  }
}

Figure 3: An Auction Contract Written in Quartz

```
3.4 Language Semantics

Here, we formally define a subset of the operational semantics of the Quartz DSL. Evaluation rules for expressions, which are generally routine, are omitted for brevity. A Quartz state machine is formally defined as a 4-tuple \((Q, q_0, T, F)\) where \(Q\) is a set of states, \(q_0\) is the initial state, \(T\) is a set of transitions, and \(F\) is a set of fields, each with a specific name and type. A transition \(t \in T\) is defined as a 7-tuple \((\text{name}, \text{src}, \text{dst}, g, a, P, B)\) where:

- \text{name}\ is the transition’s unique name.
- \text{src} and \text{dst}\ are the transition’s source and destination states.
- \(g\) is a boolean-valued guard expression.
- \(a\) is an authorization predicate.
- \(P\) is a set of transition-specific parameter values, each with a name and type.
- \(B\) is the transition’s body: a sequence of statements executed when the transition fires.

A Quartz state machine is event triggered. An external, addressable entity (either an end user or another contract) invokes a transition by submitting a message \(m = (i, t, V)\) to the state machine specifying its identity \(i\), a transition \(t\) to execute, and a (possibly empty) set \(V\) of values for \(t\)’s parameters.

The current status \(S\) of a state machine is defined as a 7-tuple \((q, \sigma, \alpha, M, s, y, C)\) where:

- \(q\) is the machine’s current state
- \(\sigma\) is a mapping from names to values representing the current context. It always includes the state machine’s fields as well as built-in values \text{sender}, \text{balance}, and \text{now}.
- \(\alpha\) is a mapping from transitions and transition parameter values to the set of identities that have authorized invocation of that transition with that particular set of parameter assignments.
- \(M\) is a queue of transition invocation messages. The machine may only read the head of the queue or remove the head of the queue. It may not inspect any other queue elements or the queue’s length.
- \(s\) indicates which statement within a transition to execute. The special statement form \(\top\) indicates that the machine has not yet started execution of any transition, while \(\bot\) indicates that a transition has just completed. Thus when executing some transition with body \(B = [b_1, \ldots, b_n]\), \(s\) will assume the sequence of values \([\top, b_1, \ldots, b_n, \bot]\).
- \(y\) is an authorization clause that must be evaluated to determine if the invoked transition may proceed, or \(\Box\) if no such determination is in progress.
- \(C\) is a stack of transitions currently in progress.

A machine’s initial status, before it executes its initial transition, is therefore \((q_0, \{ f.\text{name} \mapsto 0.f.\text{type} : f \in F\}, \emptyset, \varepsilon, \top, \Box, \varepsilon)\), where \(\varepsilon\) denotes an empty sequence and \(0_f\) denotes the zero element of type \(T_f\).

3.4.1 Transition Authorization. A subset of Quartz’s operational semantics for transition authorization is presented in Figure 4. In all rules, \(\rightarrow\) indicates small-step evaluation and \(\parallel\) indicates big-step evaluation. \(\parallel a\) more specifically denotes big-step evaluation with \(\sigma\) as the initial environment. The \(\circ\) symbol denotes concatenation, e.g., \(m \circ M\) signifies the element \(m\) prepended to the sequence \(M\).

When a new message \(m\) is dequeued, the machine’s status is updated to reflect that the message’s sender, \(i\), approves of the transition’s execution, and Quartz begins evaluation of the relevant transition’s authorization clause, \(t.a\), as shown in \text{UpdateAuth}. Terms in the authorization clause are of the forms \(i, \text{any}(I)\), and \(\text{all}(I)\). These terms are evaluated as expected, shown in \text{AuthSingleTrue}, \text{AuthAnyTrue}, and \text{AuthAllTrue}. Each of these rules has a complement, e.g., \text{AuthSingleFalse}, omitted for brevity. Authorization clauses may also use Boolean \&\& and \(|\!|\!|\) as connectives, with the expected evaluation rules.

If we have \(t.a \parallel \top\), then evaluation proceeds to the transition’s body. As shown in \text{AuthSuccess}, the current statement in the machine’s status advances from \(\top\) to \(b_1\), the first element of \(t.b\). Otherwise, Quartz discards message \(m\), removes the transition’s parameter assignments from the current context \(\sigma\), and awaits the next incoming message. However, the sender’s approval of the transition persists in \(\alpha\) and affects future evaluations of \(t.a\).

\[
S = (q, \sigma, \alpha, m \circ M, \top, \Box, C) \quad m = (i, t, V)
\]

\[
\frac{tg \parallel_\sigma \top \quad \alpha_{t,V} = \alpha(t, V)}{S \rightarrow (q', \sigma', \{ (t, V) \mapsto \alpha_{t,V} \cup \{\} \mid \alpha \circ m \circ M, \top, i, a, C)\} \quad (\text{UpdateAuth})}
\]

\[
S = (q, \sigma, \alpha, m \circ M, i, C) \quad m = (\langle i, t, V \rangle)
\]

\[
\frac{s \in \alpha(t, V)}{S \rightarrow (q, \sigma, \alpha, m \circ M, \top, \true, C)} \quad (\text{AuthSingleTrue})
\]

\[
S = (q, \sigma, \alpha, m \circ M, \top, \false, C) \quad m = (\langle i, t, V \rangle)
\]

\[
\frac{s \notin \alpha(t, V)}{S \rightarrow (q, \sigma, \alpha, m \circ M, \top, \false, C)} \quad (\text{AuthSingleFalse})
\]

\[
S = (q, \sigma, \alpha, m \circ M, \top, \true, C) \quad m = (\langle i, t, V \rangle)
\]

\[
\frac{\exists i \in I : i \in \alpha(t, V)}{S \rightarrow (q, \sigma, \alpha, m \circ M, \top, \true, C)} \quad (\text{AuthAllTrue})
\]

\[
S = (q, \sigma, \alpha, m \circ M, \top, \false, C) \quad m = (\langle i, t, V \rangle)
\]

\[
\frac{\forall i \in I : i \in \alpha(t, V)}{S \rightarrow (q, \sigma, \alpha, m \circ M, \top, \false, C)} \quad (\text{AuthAllFalse})
\]

\[
S = (q, \sigma, \alpha, m \circ M, \top, \top, C) \quad m = (\langle i, t, V \rangle)
\]

\[
\frac{\top \in \alpha(t, V)}{S \rightarrow (q, \sigma, \alpha, m \circ M, \top, \top, C)} \quad (\text{AuthSuccess})
\]

\[
S = (q, \sigma, \alpha, m \circ M, \top, \false, C) \quad m = (\langle i, t, V \rangle)
\]

\[
\frac{\top \notin \alpha(t, V)}{S \rightarrow (q, \sigma, \alpha, m \circ M, \top, \false, C)} \quad (\text{AuthFailure})
\]

Figure 4: Quartz Operational Semantics: Authorization

3.4.2 Transition Execution. A transition’s body consists of a sequence of statements, evaluated in order when the transition executes. Evaluation rules for Quartz’s small set of statement types are presented in Figure 5 and are relatively straightforward. When the end of the transition’s body, \(\bot\), is reached, Quartz consults the state machine’s stack for a record of an in-progress parent transition, consisting of its environment \(\sigma\) and the next statement to execute \(s\). If such a record exists, control returns to the parent transition. Otherwise, the state machine moves to the implicit \(\top\) statement. In both cases, the message at the head of the queue \(M\) is finally removed, the current environment \(\sigma\) is stripped of the transition’s parameter assignments, and prior approvals of the transition’s execution in \(\alpha\) are cleared.

Quartz’s \text{send} statement has slightly more complex evaluation rules. Because a \text{send} involves ceding control to the recipient, its
A well-formed state machine has a unique initial transition with description before it attempts to generate TLA due to a name collision and that all guards, authorization predicates, and statements refer to well-defined variables. Quartz description must be reachable from the initial state following some transition, expressed in , or it may reenter the state machine by invoking an arbitrary evaluation non-deterministically produces one of several possible outcomes, each shown in Figure 6. In the first case, SendSuccess completes without any disruption to control flow. Execution proceeds to the next statement. Otherwise, the recipient may either throw an exception, halting progress as denoted in SendError, or it may reenter the state machine by invoking an arbitrary transition, expressed in SendReenter. This final evaluation rule is the only means by which the machine’s stack C may grow.

\[
S = \{q, \sigma, a, M, s, \square, C\} \quad s = \text{“send a to i”} \\quad a \parallel_\alpha a' \quad d' \leq \sigma[balance] \quad b = a[balance] - a' \\
S \rightarrow \langle q, [balance \mapsto b] \sigma, a, M, \text{next(s)}, \square, C \rangle
\]

***Figure 5: Quartz Operational Semantics: Statements***

4.2 Specification Generation

Quartz’s specification generator targets PlusCal, an intermediate language built on top of the original TLA specification language. PlusCal for pieces of the case study is provided in Appendix B. PlusCal offers several features that make it a more natural target than TLA itself, such as procedures to model state transitions and conditionals to model transition guards. Translating PlusCal data types and transitions into PlusCal is straightforward,

also performs simple type checking to avoid errors in subsequent code generation phases. Much of this is as expected, for example checking that arithmetic and logical operators are only performed to compatible types. Guards must be Boolean-valued, the target of the and not in operators must be a Sequence, token sends only target expressions of Identity type, and so on.

4.1 Why Model Checking and TLA+?

We chose bounded model checking as Quartz’s core verification technique because it does not require significant intervention from the end user, i.e., the contract author. Although a contract author must write the invariants she would like to have verified, Quartz fully automates the more difficult task of writing a formal specification of the contract’s behavior and its execution environment that is suitable as input to a model checker. The model checking also offers immediate useful feedback to the user as output — an execution trace that produces a violation of one or more of the desired properties. This feedback helps guide a contract author in making refinements to her state machine.

TLA+ serves as Quartz’s target specification language and its verification backend. TLA+ and its model checker, TLC, are relatively mature, well-documented, and have been successfully applied in developing and testing significant systems [40]. More modern model checkers have since emerged, but they tend to be inherently tied to the semantics of particular implementation languages such as C [26] or operate at the low level of bytecode [24]. The flexibility of TLA+’s specification language simplifies Quartz’s task of generating a formal contract specification. This becomes especially important when describing the execution semantics of Solidity, which have important differences from the semantics of traditional programming languages. Moreover, there are ongoing efforts to modernize verification in TLA+, such as symbolic model checking with SMT solvers [32], that Quartz may be able to use in the future.
but modeling Quartz’s authorization semantics and the blockchain execution environment is more challenging. The PlusCal generated by Quartz is translated into TLA* with off-the-shelf tools.

Data Types. Every data type in Quartz maps to a counterpart in PlusCal. Many have direct equivalents such as Ints and Maps. Quartz defines the domain of the Identity type as a fixed set of symbolic constants, with a user-configurable size. The translation of a HashValue is more subtle. Quartz has no need to model hash functions in detail aside from the fact that they are assumed to be injective, nor does it require that HashValue instances are ordered. The output of hash(x₁, ..., xₙ) is simply modeled as a PlusCal tuple (x₁, ..., xₙ), preserving injectivity and enabling equality checks among HashValue instances.

Transitions. Each transition defined in a Quartz contract’s state machine is generated as a PlusCal procedure, with transition parameters naturally mapping to procedure parameters. An extra parameter is added to the PlusCal procedure to track the transition’s sender. An auxiliary field is used to track the machine’s current state. The procedure body begins with three conditional checks: one to ensure that the state machine is currently in the transition’s designated starting state, a second to ensure that the transition’s guard, if defined, is satisfied, and a third to ensure that the transition’s authorization predicate, if defined, is also satisfied. Finally, the statements forming the transition body are converted to PlusCal in the expected way, as PlusCal supports a standard collection of arithmetic and logical operators.

Authorization. Quartz adds auxiliary fields to a contract’s PlusCal specification to accurately model the authorization semantics detailed in Figure 4. Note that an entity approves of the execution of a transition for a particular set of parameter values. For example, consider a transition T with input parameters of types t₁, ..., tₜ that includes a term of the form all(I) in its authorization predicate. Quartz generates a PlusCal function (associative array) F of type tₜ₁ × ... × tₜ₄ × Identity → Boolean. Then, all(I) is evaluated in PlusCal, which has native support for quantifiers, as ∀i ∈ 1 : F(p₁, ..., pₙ, i), where pᵢ is the iⁿᵈ transition argument. Similar translations are performed as needed for the other authorization term forms. Quartz generates the minimum number of auxiliary fields, only when authorization predicates cannot be satisfied by just a single transition approval.

Modeling the Environment. Once an Ethereum contract is deployed to a blockchain, any of its transformations may be invoked at any time, by any user. For a Quartz contract, this means any of the state machine’s transitions may be invoked at any time. The PlusCal model generated by Quartz is organized around a main invocation loop that simulates this environment. Each time through the loop, a transition t is non-deterministically selected for execution. Values for its input parameters p₁, ..., pₙ are similarly selected from their respective domains, including an identity i as sender.

The second major challenge in modeling the Ethereum execution environment is capturing the behavior of sending tokens from one contract to another, i.e., the semantics of Figure 6. As explained above, there are two primary means of exchanging tokens between one Ethereum contract and another: using Solidity’s transfer primitive or using the call primitive. Both yield control to the destination contract. call is more flexible in that it allows the recipient to execute arbitrary code, but this may include a reentrant invocation of the sending contract. transfer restricts execution but propagates any exceptions thrown by the receiver back to the sender, which may block forward progress.

The user may specify the use of transfer or call as a configuration parameter. Quartz is capable of modeling either primitive’s behavior. The generated PlusCal model deducts from the balance field and then makes a non-deterministic choice to model the recipient’s response. When modeling transfer, the recipient either does nothing, indicating a routine token transfer, or throws an exception. When modeling call, the recipient either does nothing, meaning any code executed by the recipient had no consequence for the sender, or it non-deterministically selects some transition t of the sender’s to invoke, modeling possible re-entrant execution.

The behavior of Ethereum’s exceptions cannot be expressed in PlusCal. Instead, Quartz generates an initial PlusCal specification, invokes the PlusCal translator to produce TLA*, and modifies this code directly. The final TLA* generated by Quartz for model checking formalizes unwinding of the stack upon an exception: reverting the contract’s fields to their state before the current call chain and jumping to the main invocation loop to begin a fresh transition.

4.3 Bounding the Search Space
The TLA* specifications generated by Quartz, as described so far, have an infinite execution space. We must apply bounds to this search space to ensure that model checking terminates. Quartz exposes a set of parameters that the user may set at verification time. All of these parameters convey aspects of the contract’s execution or the domain of data types:

- Minimum and maximum integer values
- Number of distinct Identity instances to model
- The maximum call depth reached during transition execution
- The maximum number of iterations of the main transition invocation loop

The model checker is essentially exploring all possible sequences of state machine transitions. The first two parameters above limit the branching factor of the search space, while the third and fourth limit the depth to which it is explored.

5 SOLIDITY GENERATION
Quartz is able to translate state machine descriptions to Solidity implementations, enabling seamless deployment once a contract has been sufficiently validated by its developer. Quartz targets Solidity rather than EVM bytecode for several reasons. Solidity is more human readable than EVM bytecode, which means a contract author may easily inspect and audit a generated implementation if necessary. Moreover, there is an ongoing effort within the Ethereum community to replace the original EVM with a new virtual machine based on WebAssembly [20]. By targeting Solidity, Quartz remains agnostic to this potential change.

Data Types. The data types for state machine fields and transition parameters in Quartz each have a natural analogue in Solidity. For example, an Identity corresponds to a Solidity address. More recent versions of Solidity require address variables that are the
target of a `send` to be explicitly annotated with the payable keyword. When `Quartz` generates Solidity, it infers all necessary uses of the payable designation.

**State Transitions.** A state transition maps to a Solidity function, where transition parameters have the expected correspondence to function parameters. An auxiliary field represents the `Quartz` machine’s current state, and a `requires` statement at the beginning of the function ensures that the machine is in the proper starting state, otherwise the function (transition) does not proceed. If the transition defines a guard, it is evaluated within a second `requires` statement. Translation of the transition’s body is straightforward, as Solidity features all of the usual operators and assignment semantics. Finally, the machine’s current state is updated to the designated destination state of the transition.

**Transition Authorization.** As with PlusCal, `Quartz` augments a Solidity contract with additional fields for tracking prior authorizations when needed, modifies these fields when a transition with an authorization predicate is invoked, and checks these fields within a conditional to ensure that the authorization predicate is satisfied before a transition is executed. Unlike for guards, authorization predicates are not checked within a `requires` as this would revert the action of recording the sender’s approval for the transition. Solidity does not have the same flexibility as TLA++, and generating fields to track prior authorizations is more involved in this setting, particularly in recording authorization for each possible combination of input parameters. Given a transition with parameters of type `p1, ..., pn`, `Quartz` generates Solidity that hashes the concatenation of the parameter values and uses the result to look up prior authorizations.

`Quartz`-generated Solidity uses `mapping` instances to efficiently record and look up prior transition authorizations. However, groups of entities referred to by a `Quartz` any or all authorization term are represented as arrays, which means we must use loops for certain operations on these groups. This is because Solidity mappings do not permit iteration over their members and therefore cannot be used to emulate something like a set data structure. In addition, the `Quartz` generation logic must produce code that is as flexible as possible, with no assumptions or domain knowledge about how identity groups are used or modified over time. A `Quartz` Sequence of `Identity` values could be modified by a `clear` statement at any time, and there is no efficient way to empty a Solidity mapping.

Loops can be problematic in Solidity because of termination and gas cost issues. However, loops generated by `Quartz` are only used for simple scans of finite data structures, and as we show in our evaluation below, simple `Quartz`-generated Solidity with loops often actually has lower gas costs for practical workloads than handwritten code that uses more complex data structures to avoid them. If warranted, we could extend `Quartz` to generate code following Solidity’s iterable mapping pattern\(^1\) or to perform additional optimizations when an identity group is never cleared.

6 EVALUATION

This section presents our evaluation of `Quartz`. We begin by measuring the amount of code required to express various contracts in

---

\(^1\)https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol

---

Quartz versus in Solidity and the amount of code generated from a `Quartz` description. Next, we discuss results from the model checking process in depth for two case studies. Finally, we compare the execution costs of `Quartz`-generated and handwritten contracts.

6.1 Contract Size

We use lines of code as a proxy for code complexity and required developer effort. Here, we are interested in the extent to which `Quartz` enables concise expression of rich contract logic, particularly when compared to Solidity, the de facto standard programming language for smart contracts. We also seek to determine if `Quartz` introduces overhead by generating significantly more verbose Solidity code than would be produced by a Solidity programmer. This is particularly relevant in the blockchain setting, where the size of a contract’s compiled bytecode directly impacts the gas costs of deploying the contract to the ledger. Finally, we quantify developer effort saved when using `Quartz` for model checking by measuring the size of a contract’s TLA\(^+\) representation.

Table 2 shows the lines of code needed to express all of our case studies in `Quartz` and in Solidity. It also shows the lines of Solidity and TLA\(^+\) generated from the `Quartz` implementation of each contract. Each row in the table corresponds to one of the case studies presented in Table 1. When writing Solidity for each case study, we modified an existing code base (cited in Table 1) whenever possible rather than starting from scratch to ensure we produced idiomatic Solidity code. We were also careful to remove any comments or extraneous lines of code, like redundant getter functions that have no `Quartz` equivalent, to make the comparison as fair as possible.

As shown in the table, every case study’s implementation in `Quartz` involved fewer lines of code than the handwritten Solidity equivalent, with the exception of the DAO, where the two are equal. The ratio of lines of `Quartz` to lines of handwritten Solidity is 0.68 on average and as small as 0.34, for the `StaticMultiSig` case study. `Quartz` was particularly concise for the multi-sig wallets and `RockPaperScissors` because of its state machine structure and authorization predicates. These allow a `Quartz` developer to avoid writing tedious and repetitive assertions at the beginning of contract transactions to verify that the contract is in the expected state before proceeding.

`Quartz` typically, but not always, generates Solidity code that is more verbose than the handwritten equivalent. There are cases where the generated code is shorter than the handwritten code. This is often the case when the handwritten Solidity leverages domain knowledge to use more verbose but arguably more efficient data structures, particularly to track authorization, than the general code produced by `Quartz`.

Finally, significantly more TLA\(^+\) code is needed to express each contract than Solidity or `Quartz` code. This is mainly because a contract’s TLA\(^+\) specification expresses both the contract’s logic and its execution semantics. In particular, the TLA\(^+\) representation of any contract must describe the main invocation loop in which any user may invoke any of the contract’s transitions with arbitrary parameter values. It must also express the potential for reentrant execution after a `send` and exception handling. Therefore, even a short contract generates a relatively lengthy TLA\(^+\) specification.
6.2 Model Checking

Here, we describe experiences model checking two contracts using Quartz. For both contracts, Quartz helps to surface non-obvious bugs that could easily be overlooked during contract development. Below, we report the time required for model checking to find invariant violations. These times were obtained on a workstation with an Intel i7-6700 CPU and 32 GiB of RAM running version 2.13 of the TLC model checker with 8 worker threads.

6.2.1 Model Checking an Auction Implementation. We introduced the Auction case study in detail in Section 3.2. Its state machine appears in Figure 2 and Quartz code appears in Figure 3. While it may appear perfectly logical, the contract as presented above features multiple security vulnerabilities, related to it distribution of payments back to surpassed bidders and to the seller. These vulnerabilities are particularly insidious because they emerge from code that appears innocuous. They are good examples of how Ethereum’s execution semantics differ from those of traditional software and can trip up contract authors.

To begin, consider the following invariant for the auction:

\[ p_1 : \text{Closed} \Rightarrow \text{HighestBid} \geq \max(\text{submitBid}.\text{tokens}) \]

This property takes advantage of a number of Quartz’s features for writing invariants. It states that if the auction reaches the Closed state, then the value of HighestBid should be greater than or equal to the maximum value ever assigned to the tokens parameter for the submitBid transition.

Recall that Quartz may generate a contract that uses either Ethereum’s transfer construct or the call construct for dispensing currency. The choice is configurable by the user, and Quartz is fully capable of generating TLA\(^+\) to model either. If transfer is used, the model checker finds the following violation of \( p_1 \). This required only 2 seconds to complete on our test system.

(1) Identity \( I_1 \) deploys a new auction contract. The auction enters the Init state.

(2) \( I_2 \) submits an initial bid of 2 tokens and is recorded as the highest bidder. The auction enters the Open state.

(3) \( I_3 \) submits a new bid of 4 tokens. The auction sends 2 tokens back to \( I_2 \) as a refund, since they are no longer the highest bidder.

(4) \( I_2 \) reacts by throwing an exception. This propagates back to the auction contract (due to the use of transfer) and the current transition is aborted. \( I_1 \)'s bid is lost.

(5) No additional bids are submitted before \( I_1 \) moves to close the auction and \( I_3 \) is declared the winner.

Here, \( I_2 \) is able to hijack the auction and prevent itself from being supplanted as the highest bidder, rigging the results of the auction. \( p_1 \) is also violated if we use call rather than transfer, again because of an issue in refunding a previous bidder. TLC found the following trace in 6 seconds on our test system.

(1) Identity \( I_1 \) deploys a new auction contract. The auction enters the Init state.

(2) \( I_2 \) submits an initial bid of 2 tokens and is recorded as the highest bidder. The auction enters the Open state.

(3) \( I_3 \) submits a new bid of 3 tokens. The auction sends 2 tokens to \( I_2 \) as a refund.

(4) \( I_2 \) responds to the receipt of tokens by submitting a new bid of its own, with a value of 4 tokens, creating a reentrant invocation of the submitBid transition.

(5) The auction accepts \( I_2 \)'s bid, sets HighestBidder to \( I_2 \) and HighestBid to 4.

(6) Control returns to the parent transition, which has just completed its send. It updates HighestBidder to \( I_3 \) and HighestBid to 3, accounting for \( I_3 \)'s bid but overwriting \( I_2 \)'s bid.

(7) No further bids arrive. The auction reaches the Closed state.

Here, we see that contract re-entrancy, an issue best known for its exploitation by malicious actors, can also lead to undesirable outcomes for well-intentioned actors. One could easily imagine a developer seeking to create a contract that submits a bid on her behalf in reaction to having just been displaced as an auction’s winner, possibly to implement some bidding strategy, yet that would go awry in this implementation.

To address either of these bugs, the contract author could instead store a Quartz Map[Identity, Uint] tracking pending refunds that is updated when a newly winning bid is submitted. A previous bidder must invoke a separate transition to ask the contract to send her a refund, decoupling this from bidding. This is a well-known design pattern in Solidity [17], although it is prone to its own re-entrancy issues, which Quartz can also identify through its verification. Say we modify submitBid accordingly and add the following transition to allow bidders to claim refunds once the seller has redeemed their winnings:

```solidity
refund: redeemed -> redeemed
requires [ Balances[sender] > 0 ] {
    send Balances[sender] to sender
    Balances[sender] = 0
}
```

Consider the following new invariant for the auction:

\[ p_2 : \text{balance} \geq 0 \]
This states that the contract’s balance cannot go negative, i.e., it cannot dispense more tokens than it receives. While this is impossible for a contract running on the Ethereum blockchain, negative contract balances are within the search space defined by Quartz for model checking because they can usefully indicate a contract’s vulnerability to unbounded token withdrawals. Indeed, if Quartz is configured to model behavior of token sends using call, model checking finds the following violation in 15 seconds:

1. Identity \( I_1 \) deploys a new auction contract.
2. \( I_2 \) submits an initial bid of 1 token. The auction enters the Open state with balance = 1.
3. \( I_3 \) submits a new bid of 4 tokens, hence balance = 5.
4. No subsequent bids are submitted before the deadline, and \( I_1 \) moves to close the auction. The auction enters the Closed state.
5. \( I_1 \) invokes the redeem transition, receiving its winnings. Now, balance = 1 and the auction enters the Redeemed state.
6. \( I_2 \) invokes the refund transition and is sent the 1 token recorded in Balances\([I_2]\]. Now, balance = 0.
7. In reaction to this receipt of tokens, \( I_2 \) makes a reentrant invocation of refund. Balances\([\text{sender}]\) has not yet been updated in the parent transition, so the child transition’s guard is satisfied.
8. Another send of 1 token to \( I_2 \) is attempted, and balance = -1, violating the invariant.

This execution trace illustrates the fundamental vulnerability behind the famous compromise of the DAO contract [13]. The usual advice to Solidity developers is to set a temporary variable to the amount of tokens to send, then subtract from the appropriate contract field before executing a send referencing the temporary variable. Quartz offers an alternative sendAndConsume construct that will generate such code.

### 6.2.2 Model Checking ERC-1540

Quartz is useful not just for identifying subtle consequences of a contract’s execution semantics, but also for identifying more routine logic errors that occur during the development process. Unlike our auction contract, which we initially developed as a litmus test for Quartz’s ability to surface reentrancy and exception issues, we drafted an initial implementation of ERC-1540 after Quartz was fairly mature, chose an invariant to verify, and used Quartz to refine the contract.

ERC-1540 is a proposed Ethereum standard interface for an asset management contract. Among other capabilities, it allows an owner to sell shares, issue dividends, or transfer control of the asset, all of which is tracked on the blockchain. Investors issue transactions against the contract to buy and sell shares. The Quartz implementation of ERC-1540 is considerably more complex than the auction seen previously. It uses five states and 16 transitions. The portion of the state machine relevant for the following discussion is shown in Figure 7.

When the contract is initialized, its creator is recognized as the asset’s owner. It begins in the Unissued state, meaning there are no outstanding shares. If this is the case, the owner is free to transfer possession of the asset to another party, as shown in the transition at the top of Figure 7. The owner may choose to move the asset to the Issued state by enacting the release of a fixed number of shares. The contract features additional transitions to exchange shares not shown in the figure. If any single party accumulates all outstanding shares, they are allowed to declare themselves as the new owner and convert the asset back to the Unissued state.

We tested our initial version of the contract by verifying the following invariant.

\[
p_1 : \text{sum}(\text{Shares}) = \text{TotalShares} \tag{1}
\]

That is, we wanted to verify that all of the asset’s shares are properly conserved as they change hands.

After generating a TLA* specification and running it through TLC, we were informed that many of the contract’s states, including Issued were not reachable. This was because we failed to properly initialize the contract’s owner field in the state machine’s initial transition. We were also able to uncover an error in our arithmetic when transferring shares.

More interestingly, we initially forgot to update the asset’s shares when an entity converts it from Issued to Unissued. Initially, we simply reassigned the owner field to the transition’s sender as shown in the figure, forgetting to zero out TotalShares and Shares\([\text{sender}]\). This enables the following trace, identified by TLC when given a Quartz-generated contract spec and simplified for brevity:

1. \( I_1 \) deploys a new ERC-1540 contract and is recorded as the owner.
2. \( I_1 \) issues 2 shares for the asset, initially owning both of them.
   - The asset enters the Issued state, and we have TotalShares = 2 and Shares\([I_1]\) = 2.
3. \( I_1 \) converts the asset back Unissued.
4. \( I_1 \) transfers ownership to \( I_2 \).
5. \( I_2 \) decides to issue 3 shares for the asset, all initially assigned to itself. In the transition body, we set Shares\([I_2]\) = 3 and TotalShares = 3. However, we still have Shares\([I_1]\) = 2. Thus, sum(Shares) ≠ TotalShares.

The solution is to add two lines to the transition from Issued to Unissued, Shares\([\text{sender}]\) = 0 and TotalShares = 0, to properly reflect the fact that the asset no longer has shares. Because ERC-1540 is more complex than the auction above, model checking takes longer. This trace was produced after TLC ran for 4 minutes on our test workstation, while the arithmetic error in share transfers required 27 minutes to find. Running times of this magnitude are not atypical for model checking.
6.3 Execution Overhead

Finally, we measured the execution efficiency of Solidity contracts generated from Quartz descriptions and those of equivalent handwritten Solidity contracts. The handwritten contracts are the same as those described in Section 6.1, meaning they are adapted from existing codebases when available and simplified if necessary, e.g., by removing extra getter functions, to form a fair comparison.

Execution of Ethereum contracts is metered by gas, a cost assigned to each virtual machine operation, to ensure termination and discourage unnecessarily expensive contract code. It is therefore natural to measure a contract’s execution efficiency by the gas it consumes. To accomplish this, we deployed both generated and handwritten versions of all case study contracts to a small private blockchain backed by nodes hosted on Amazon EC2 virtual machines. All members of the network ran Geth version 1.8.26 and used Geth’s Clique proof-of-authority consensus mechanism. This allowed us to configure the network to use a fixed gasPrice. As a result, the gas cost of a particular workload is deterministic and reproducible. It does not fluctuate with network load as it would in a proof-of-work Ethereum network.

We wrote a contract client script for each case study using Python’s Web3 library. Each script deploys the generated and handwritten versions of Solidity code, invokes an equivalent sequence of transactions against both versions, and tallies all gas costs. For example, the script for the Auction case study initializes each contract and submits the same sequence of bids to both. The results of these measurements are shown in Figure 8. Each case study is represented along the x axis by a pair of bars. The number above each pair is the ratio of total gas costs for the generated Solidity code to total gas costs for the handwritten equivalent.

Gas costs for Quartz-generated Solidity contracts are competitive with those of handwritten contracts. While the overhead is 53% for the simple ERC-1638 contract, for more substantial contracts it never exceeds 20%. Interestingly, there are some case studies where the generated contract actually has lower gas costs than the handwritten equivalent. Upon further investigation, we found that this was usually due to Quartz’s use of fewer, simpler data structures in its generated code. This typically gave the generated contract a cheaper constructor and, for some case studies, cheaper transactions when operating on a smaller body of state.

The multi-signature wallets are a good example. The handwritten wallets are based on a design used in production by Parity [44] and OpenZeppelin [43] where approvals by designated signers are tracked with both a Solidity mapping instance, to emulate a set and path but also requires bookkeeping to manage both the mapping and array. Quartz takes the simpler approach of using an array of signers and loops to check if enough signers have approved a transaction. This makes the Quartz wallets’ constructors cheaper (there are fewer fields and less bytecode) and makes transactions cheaper when the total number of signers is small. The advantage of the Quartz wallets decreases under workloads with more signers.

The disadvantage of the Quartz approach is its use of loops, which means gas costs for wallet transactions grow with the number of signers. The advantage, however, is that the code generated by Quartz is flexible, because it must accommodate any valid sequence of Quartz operations against the group of signers and authorization checks against it, i.e., it cannot exploit domain knowledge and optimize based on assumptions of how authorized signers are added or removed over time.

7 RELATED WORK

Smart contracts have attracted immense interest in both industry and academia, making them a popular target for language design and formal methods. Additionally, software development and verification based on state machines has a long history with many interesting applications. We summarize some of the most significant related works below.

7.1 State Machine-Based Development

Quartz is most directly inspired by prior works that similarly leverage a programming abstraction based on state machines to facilitate development and systematic testing of critical software. Teapot [10] is a domain-specific language used to write state machine implementations of the cache coherence protocol. It allows the programmer to translate their state machine into a specification for the Murphi model checker and into an implementation in C. Transit [57] takes this a step further, allowing the programmer to partially implement a distributed protocol as a state machine, then synthesizing the rest of the implementation using a counter-example guided inductive synthesis (CEGIS) loop. One could imagine extending Quartz to synthesize contract code using TLC in a similar approach.

Quartz is also similar to P [15], a domain-specific language for event-driven programming. Programs in P are written as state machines which can then be model checked and converted into implementations expressed in C. However, P expects the developer to formalize the state machine’s environment by writing a second “ghost” state machine. Quartz has no similar expectation. It generates TLA+ code that formalizes blockchain execution semantics.

There are also works that apply state machines to smart contract development. In FSolidM [36], a user builds a contract state machine in a graphical editor, but there is no effort at verification, and the user must manually write Solidity to complete the contract. VeriSolid [37] extended FSolidM with verification, but the user must still manually write Solidity, and VeriSolid can only reason about limited properties. Finally, Obsidian [11, 12] is a contract programming language that also features state machines as the primary abstraction. It does not involve any verification, but rather uses language features like linear types to guarantee certain properties.

7.2 Contract Programming Languages

Smart contracts have become a very popular domain for new programming languages, particularly as shortcomings to Solidity [18] have emerged. Most of these are fully-featured programming languages [1, 9, 19, 28, 47, 48], without associated tools to validate contract properties. Tezos has introduced a high-level programming language [53] that compiles to a stack-based language [52] designed to be amenable to formal analysis. However, this analysis would not be automated, but rather requires manual use of a theorem prover like Coq.
There are also some examples of more simplified, domain-specific contract languages. Frantz and Nowostawski [22] built a tool in which developers express the operations and rules for use of a contract in a human-readable DSL. This contract description is then translated into a Solidity template that the developer must manually complete. Findel [6] is a DSL specifically for expressing financial derivative contracts, heavily inspired by prior work on similar DSLs [46]. Finally, Cardano has built Marlowe [29], a Haskell-embedded DSL for financial contracts.

Quartz has similarities to Scilla [50], a functional style intermediate language for smart contracts. A Scilla contract is also written as a state machine. Unlike Quartz, Scilla is a fully-featured programming language rather than a DSL, and therefore tends to be less concise. Its authors report that ERC-20 and ERC-721 respectively require 158 and 270 lines of code, where Quartz requires 43 and 48 lines, respectively. Finally, Scilla does support automated contract analysis, but only through a library of pre-written static analyzers that must generalize to any contract. Unlike in Quartz, checking functional properties specific to a contract remains manual.

7.3 Contract Analysis Tools

While there have been several one-off efforts to verify the properties of a single contract [5, 39], many reusable tools have been built to analyze the behavior and identify vulnerabilities of existing contracts, most commonly by applying symbolic execution techniques to their EVM bytecode representations [33, 35, 38, 42] or static analysis of the contract’s AST [54]. These tools offer fully automated analysis, but they are largely restricted to identifying a fixed collection of generic vulnerabilities. Ztus [31] is a tool to convert a Solidity contract into LLVM bytecode for model checking. Securify [56] analyzes EVM bytecode to extract control flow and data flow graphs. It then verifies contract properties written in a datalog-based DSL, although these properties are intended to be generalized across many contracts rather than customized to a specific contract as in Quartz.

VerX [45] is a verification tool for Ethereum contracts that uses predicate abstraction and symbolic execution of EVM bytecode. Like Quartz, it offers automated verification of contract-specific properties, written in a variant of linear temporal logic. This makes VerX arguably the most comparable analysis tool to Quartz. However, VerX makes the assumption that all of the contracts it analyzes are effectively callback free [23], meaning it cannot model something like a token send may disrupt a contract, either through reentrancy or exceptions. Quartz does not have this restriction, meaning it could find violations to properties \( p_1 \) and \( p_2 \) discussed in Section 6.2 that VerX cannot.

8 CONCLUSION

This paper presented Quartz, a tool for implementing and testing secure smart contracts, using state machines as its fundamental organizing principle. Quartz offers developers a small, specialized language for writing contract logic with features motivated by real smart contract applications, such as transition authorization as a first-class primitive. Quartz also supports validation of a contract’s properties through translation into a TLA+ representation suitable for model checking. Once validated, a developer may seamlessly deploy her contract to a blockchain by using Quartz to generate Solidity. Our evaluation demonstrates that Quartz contracts are more concise than their handwritten Solidity counterparts. We presented two in-depth case studies of contract validation in which Quartz surfaces significant contract vulnerabilities and offers helpful execution traces that assist the developer in patching these vulnerabilities. Finally, we have shown that Quartz imposes only modest overhead in terms of both the size and execution efficiency of its generated contract implementations.

We believe Quartz can serve as a platform for true smart contract engineering, replacing the more ad-hoc development and testing processes currently in use. Even well-crafted unit test suites may overlook critical contract vulnerabilities that only arise as the result of specific, unanticipated sequences of events. Through Quartz-facilitated model checking, such execution traces are revealed and
used to inform refinements to the contract’s code that harden it against attack. An implementation is then generated and deployed for production use against real users and potential adversaries only once the contract is validated with respect to the desired properties.

There are a number of potential directions for future work. First, while Quartz state machines may currently only communicate by sending tokens, we plan to extend the DSL to support invocation of transitions in external state machines. This would require changes to TLA\(^*\) specification generation, but would allow developers to create blockchain applications from compositions of Quartz state machines. Additionally, we are considering targeting additional smart contract platforms, such as Hyperledger Fabric [2], for the deployment of Quartz contracts. This would involve both targeting a new contract implementation language, such as Go, and properly formalizing the execution semantics of the new platform in TLA\(^*\).

REFERENCES


[2] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure Validation of decentralised smart contracts through game theory and formal specification generation, but would allow developers to create blockchain applications from compositions of Quartz state machines. Additionally, we are considering targeting additional smart contract platforms, such as Hyperledger Fabric [2], for the deployment of Quartz contracts. This would involve both targeting a new contract implementation language, such as Go, and properly formalizing the execution semantics of the new platform in TLA\(^*\).
Symposium on Security and Privacy. 17.


A LANGUAGE SYNTAX

The syntax for Quartz is formally defined below. Syntax for literals as well as for arithmetic and logical operators is as expected and omitted for brevity.

\[
\text{contract} ::= \text{contract} \text{name } ( ( \text{structDecl })* \text{fields }) \text{ transition } \} \}
\]

\[
\text{data} ::= \text{data } ( ( \text{field} ) ) \}
\]

\[
\text{structDecl} ::= \text{struct } \text{structName } ( ( \text{field} ) ) \}
\]

\[
\text{transition} ::= \text{name } ( ( \text{sourceSt } ) \rightarrow\rightarrow* \text{params } \text{dest } \text{authPred } \text{guard } ) \}
\]

\[
\text{structBody} ::= \{ \}
\]

\[
\text{field} ::= \text{name } ( \text{type} )
\]

\[
\text{type} ::= \text{Int } | \text{Timestamp } | \text{Timespan } | \text{Bool } | \text{Map } ( \text{type } ) | \text{Sequence } ( \text{type } ) | \text{HashValue } ( \text{typeList } )
\]

\[
\text{properties} ::= \{ \}
\]

\[
\text{propertySpec} ::= \text{properties } ( ( \text{expr} ) )
\]

\[
\text{sourceSt} ::= \text{expr } ( \text{source} )
\]

\[
\text{guard} ::= \text{expr } ( \text{requires } ( ( (\text{type} ) ) )
\]

\[
\text{authPred} ::= \text{expr } ( \text{authorized } ( (\text{type} ) )
\]

\[
\text{transBody} ::= \text{stmt } ( (\text{stmt} ) )
\]

\[
\text{stmt} ::= \{ \text{expr } ( (\text{expr} ) )
\]

\[
\{\text{mapRef} \} ::= \\{\text{value } ( (\text{expr} ) )
\}
\]

\[
\{\text{structRef} \} ::= \\{\text{value } ( (\text{expr} )
\]

\[
\text{expr} ::= \text{balance } \text{sender } \text{now } ( \text{expr} ) \}
\]

\[
\text{max } ( (\text{expr} ) ) \}
\]

\[
\text{hash } ( (\text{expr} )
\]

\[
\text{binOp } ( (\text{expr} )
\]

\[
\text{any } ( (\text{expr} ) \}
\]

\[
\text{append } ( (\text{expr} ) \}
\]

\[
\{\text{clearSeq} \} ::= \{\text{properties } ( (\text{expr} ) )
\]

\[
\text{call } \text{initialize } ( (\text{expr} )
\]

\[
\text{call } \text{redeem } ( (\text{expr} )
\]

\[
\text{balance } ::= \text{balance } ( (\text{expr} )
\]

\[
\text{HighestBid } ::= \text{bid } ( (\text{expr} )
\]

\[
\text{end if}
\]

\[
\text{begin}
\]

\[
\text{end with}
\]

\[
\text{either}
\]

\[
\text{begin}
\]

\[
\text{end with}
\]

\[
\text{begin}
\]

\[
\text{end with}
\]

\[
\text{begin}
\]

\[
\text{end if}
\]

\[
\text{balance } := \text{balance } + \text{bid}
\]

\[
\text{HighestBid } := \text{bid}
\]

\[
\text{HighestBid } := \text{bid}
\]

\[
\text{HighestBid } := \text{bid}
\]

\[
\text{HighestBid } := \text{bid}
\]
HighestBidder := sender;
return;
end procedure;