Certifying Zero-Knowledge Circuits with Refinement Types

Junrui Liu
University of California, Santa Barbara
junrui@cs.ucsb.edu

Ian Kretz
The University of Texas at Austin
kretz@cs.utexas.edu

Hanzhi Liu
University of California, Santa Barbara
Veridise Inc.
hanzhi@ucsb.edu

Bryan Tan
Veridise Inc.
bryan@veridise.com

Jonathan Wang
Axiom
jonathanpwang@protonmail.com

Yi Sun
Axiom
yi.sun@post.harvard.edu

Luke Pearson
Polychain Capital
luke@polychain.capital

Anders Miltner
Simon Fraser University
miltner@cs.sfu.ca

Işıl Dillig
The University of Texas at Austin
Veridise Inc.
isil@cs.utexas.edu

Yu Feng
University of California, Santa Barbara
Veridise Inc.
yufeng@cs.ucsb.edu

***Abstract*—Zero-knowledge (ZK) proof systems have emerged as a promising solution for building security-sensitive applications. However, bugs in ZK applications are extremely difficult to detect and can allow a malicious party to *silently* exploit the system without leaving any observable trace. This paper presents CODA, a novel statically-typed language for building zero-knowledge applications. Critically, CODA makes it possible to *formally specify* and *statically check* properties of a ZK application through a rich *refinement type* system. One of the key challenges in formally verifying ZK applications is that they require reasoning about polynomial equations over large prime fields that go beyond the capabilities of automated theorem provers. CODA mitigates this challenge by generating a set of Coq lemmas that can be proven in an interactive manner with the help of a tactic library. We have used CODA to re-implement 79 arithmetic circuits from widely-used Circom libraries and applications. Our evaluation shows that CODA makes it possible to specify important and formally verify correctness properties of these circuits. Our evaluation also revealed 6 previously-unknown vulnerabilities in the original Circom projects.**

1. Introduction

Zero-knowledge (ZK) proof systems [20] have emerged as a promising solution for building security-sensitive applications. Because ZK proof systems allow users to prove that they know a secret without actually revealing what that secret is, they have found numerous use-cases in the context of blockchain technology. For example, privacy-protecting digital currencies like ZCash utilize zero-knowledge proofs, as do layer-2 blockchain scaling solutions known as ZK-rollups [24], [34], [31].

The goal of a zero-knowledge proof system is to generate two entities, namely a *prover* and a *verifier*. Given an input I chosen by the verifier, the goal of the prover is to construct a proof that they know a secret W satisfying a relation $R(I, W)$. In most ZK proof systems, the relation R is encoded as an *arithmetic circuit*, which is a set of polynomial equations over a finite field. However, because manually constructing such arithmetic circuits is extremely difficult and error-prone, domain-specific languages like Circom [7] try to facilitate this process by generating most of the arithmetic circuit automatically from the user-specified computation. But, even despite compiler support, building *correct* zero-knowledge applications remains a significant challenge. Indeed, several recent attacks [39], [4], [16] have shown that bugs in zk applications can allow attackers to construct bogus proofs that are accepted by the verifier. For example, a vulnerability in the ZCash protocol [16] could enable the generation of counterfeit coins, and a recent vulnerability [42] in the circom-pairing library could have allowed attackers to forge signatures.

A particularly problematic aspect of bugs in ZK applications is that they are extremely difficult to detect, *even once they are exploited*. Unlike other common types of attacks like denial-of-service or privilege escalation, ZK attacks allow a malicious party to *silently* exploit the system, without leaving any observable trace. Hence, traditional defense mechanisms, such as run-time monitoring or sandboxing, are completely powerless at mitigating attacks caused by bugs in ZK proof system implementations. This leaves *static analysis & verification* as the only viable mechanism for preventing silent attacks on ZK proof systems.

Motivated by this observation, this paper presents a new statically-typed language called CODA for building ZK applications. Critically, CODA makes it possible to *formally specify* and *statically check* properties of a ZK application through a rich *refinement type* system. Intuitively, the refinement type specifications in CODA make it possible to express the correspondence between the *computation* that the programmer has in mind and the actual *arithmetic circuit* that underlies the zero-knowledge proof system. As a result, refinement types serve as a bridge between the developers' mental computational model and the mathematical objects that are used to realize that computation in a zero-knowledge way.

At a high level, CODA serves the same role as a domain-specific language like Circom, in that it is intended to facilitate the construction of arithmetic circuits. However, unlike Circom which is a low-level imperative language, CODA is a functional language with higher-order combinators. Beyond making it possible to express the same computation in a more concise and declarative way, CODA is designed to facilitate static type checking. In particular, given the refinement type specifications annotated by the user, the CODA compiler generates a set of logical formulas whose validity entails the correctness of the circuit. That is, if the formulas generated by the CODA compiler can be proven to be logically valid, this constitutes a proof that the program conforms to its specification. Thus, type checking in CODA essentially amounts to formal verification.

One of the key challenges in formally verifying ZK applications is that they require reasoning about polynomial equations over large prime fields. This fact is important because current automated theorem provers (in particular, SMT solvers) do not provide adequate support for logical reasoning in such a theory. Hence, in contrast to recent work on *logically qualified data types* [32], [33], [40], [41], [36] that discharges typing constraints via an SMT solver, doing so is unfortunately not an option for the CODA compiler. To deal with this challenge, CODA outputs a set of lemmas in Coq [23] that can be proven in an interactive manner. Because many of the lemmas output by the CODA type checker correspond to complex number-theoretic theorems, it is unrealistic to expect that they can be discharged completely automatically. Thus, while human involvement in the type checking process is unavoidable in some cases, CODA tries to reduce the manual proof burden by providing tactics to codify common proof patterns in this domain.

We have used CODA to re-implement and formally verify 79 complex arithmetic circuits from 9 Circom projects. Out of the 79 circuits under consideration, we were able to use CODA to formally verify 61 circuits and uncovered 6 previously unknown vulnerabilities in popular Circom projects.

To summarize, this paper makes the following key contributions:

- We introduce CODA, a statically-typed functional DSL for facilitating the construction of arithmetic circuits and their compilation down to R1CS constraints [11].
- We describe a refinement type language for specifying functional correctness properties of ZK applications and present type checking rules that are formally proven to be sound. In other words, we show that type checking in CODA corresponds to formal verification.

```
template BigLessThan(k){
    signal input a[k];
    signal input b[k];
    signal output out;
    component lt[k];
    component eq[k];
    for (var i = 0; i < k; i++) {
        lt[i] = LessThan();
        lt[i].in[0] === a[i];
        lt[i].in[1] === b[i];
        eq[i] = IsEqual();
        eq[i].in[0] === a[i];
        eq[i].in[1] === b[i];
    }
    component ors[k - 1];
    component ands[k - 1];
    component eq_ands[k - 1];
    for (var i = 0; i <= k - 2; i++) {
        ands[i] = AND();
        eq_ands[i] = AND();
        ors[i] = OR();
        if (i == 0) {
           ands[i].a === eq[1].out;
           ands[i].b === lt[0].out;
          eq_ands[i].a === eq[1].out;
           eq_ands[i].b === eq[0].out;
           ors[i].a === lt[1].out;
           ors[i].b === ands[i].out;
        } else {
           // Bug! The next line should be:
           // ands[i].a === eq_ands[i - 1].out
           ands[i].a === ands[i - 1].out;
           ands[i].b === lt[i].out;
           eq_ands[i].a === eq_ands[i - 1].out
               ;
           eq_ands[i].b === eq[i].out;
           ors[i].a === ors[i - 1].out;
           ors[i].b === ands[i].out;
        }
     }
     out === ors[k-1].out;
 }
```

Figure 1. A circom program for expressing the less-than relation between a and b. The out signal is expected to be a binary field element that is equal to 1 iff a represents a smaller integer than b.

- We re-implement 79 widely-used ZK circuits from popular Circom projects in CODA and show that CODA is useful for specifying and formally verifying these circuits. Furthermore, our verification effort uncovered previously unknown vulnerabilities.

2. Overview

In this section, we motivate our proposed approach with the aid of a motivating example. To this end, we first present an existing circuit written in Circom and then explain the CODA workflow.

2.1. Motivating Example

Figure 1 shows a function called **BigLessThan** implemented in Circom. This function takes two arrays **a** and **b** representing two integers in Big Endian notation. For a specific value of **k** (representing the number of bits), the Circom compiler generates an arithmetic circuit (i.e., a polynomial equation over a finite field) such that the (binary) output signal called **out** is 1 iff the integer represented by **a** is less than that represented by **b**, and 0 otherwise. While *computing* the correct value of **out** may not seem particularly challenging, what makes this task tricky is that we actually need to *generate a set of constraints* over the signals **out**, **a**, and **b** satisfying a certain property. If the generated set of constraints is incorrect, a malicious party can engineer bogus proofs that are accepted by the verifier [42].

For the purposes of this paper, it is not necessary to understand the Circom code shown in Figure 1 except to realize that the Circom code is actually quite involved despite being based on the following simple observations:

- **Observation 1:** The length-i prefix of **a** is less than the length-i prefix of **b** if the length-$(i-1)$ prefix of **a** is less than the length-$(i-1)$ prefix of **b**, or if the length-$(i-1)$ prefixes are equal but $a[i] < b[i]$.
- **Observation 2:** The length-i prefix of **a** is equal to the length-i prefix of **b** if the length-$(i-1)$ prefixes are equal and $a[i] = b[i]$.

Unfortunately, the actual Circom code generating the constraints (using the **===** operator) is quite non-trivial and uses the **lt** and **eq** *components* (which are similar to library functions) to help implement this logic. While understanding the exact details of the Circom code is not necessary for this paper, the key take-away is that the code is far from trivial, and, in fact, contains a subtle bug at line 32, with the appropriate fix indicated in comments immediately above. Such a bug can allow a malicious prover to construct the input arrays **a** and **b** in such a way that the generated constraints will be satisfied even though the integer represented by **a** is larger than that represented by **b**.

The bug in this example may look relatively innocuous, but the consequences of such a mistake can be quite catastrophic in the context of a ZK application. For example, if this circuit was used in a DeFi application to ensure that parties cannot transfer more money than they have, an attacker could exploit this bug to transfer money than they actually own. Furthermore, in such a privacy-centric DeFi application based on a ZK proof system, it would be very difficult to detect that something bad is happening: since no one except the attacker knows exactly how much money they own, no party can detect that money is being stolen if the verifier itself is buggy.

2.2. Our approach

Figure 2 shows the implementation of the **BigLessThan** function in CODA. Even though the CODA code in Figure 2 compiles down to the same R1CS constraint representation as the Circom code from Figure 1, the CODA program is much more concise and declarative and allows the user to directly implement **Observation 1** and **Observation 2** without being bogged down in low-level, orthogonal details. In particular, the implementation of **BigLessThan** in CODA uses the functional **iter** construct of the form:

$$\mathsf{iter}_\tau\ e_s\ e_e\ e_f\ e_a$$

where e_s, e_e denote the start and end indices of the iterator, e_f is a lambda abstraction that is repeatedly applied to the accumulator, and e_a is the initial value of the accumulator. Hence, the code in Figure 2 iterates from 0 to $k-1$ (inclusive) and repeatedly applies e_f to the accumulator, which is initialized to the value $(0, 1)$. In this case, the accumulator is a pair of booleans **(lt, eq)** where **lt** (resp. **eq**) indicates whether the length i prefix of a is less than (resp. equal to) that of b. Observe that the new value of **lt** is calculated as:

$$lt \vee (eq \wedge (a[i] < b[i])$$

which corresponds directly to **Observation 1** from earlier. Similarly, the computation of **eq** also directly follows **Observation 2**. Hence, the CODA implementation of **BigLessThan** allows the developer to implement the core logic in a relatively straightforward way by *composing* existing circuits that implement boolean disjunction (#Or), conjunction (#And), and field element comparison (#LessThan).

```
circuit BigLessThan
  (k: {Z | 0 <= v})
  (a: {F | binary v}^k)
  (b: {F | binary v}^k)
  -> {F | v = (|a| < |b|) } {
    let (lt, _) = iter 0 (k-1) (
      \i. \(lt, eq). (
        #Or lt (#And eq (#LessThan a[i] b[i])),
        #And eq (#Eq a[i] b[i]))
      )
      (0, 1)
      inv:(\i.
        {F | v = (|a[:i]| < |b[:i]|)} *
        {F | v = (|a[:i]| = |b[:i]|)})
    in lt
}
```

Figure 2. A CODA DSL program for the same function in Figure 1.

Specifying circuit behavior. Beyond making it significantly easier to construct arithmetic circuits, a key aspect of the CODA DSL is that it allows *specifying* important properties of the circuit as refinement types, which are indicated in blue in Figure 2. As we can see from the type annotation on the input **k**, refinement types have the form $\{T \mid \phi\}$ where T is a base type and ϕ is a *logical qualifier*. In this case, the base type of **k** is an integer, denoted as **Z**, and the logical qualifier $0 \leq \nu$ further states that the value of **k** is non-negative. The type annotations on **a** and **b** are of the form τ^k, indicating that they are arrays of size k with elements of type τ. In this case, the element types of **a** and **b** are field elements (indicated as **F**), and the logical qualifier further restricts their values to be binary.

The most interesting part of the specification is the type annotation for the return value of **BigLessThan**, which has the following refinement type:

$$\{\mathbf{F} \mid \nu = (|a| < |b|)\}$$

This states that the return value of the function is a binary field element which is 1 iff the integer represented by **a** is less than the one represented by **b**, where the operator $|x|$

interprets an array as a base-2 big integer. [1] We note that the expression $|a| < |b|$ is valid in CODA's type system, but it is fundamentally *not* a valid *program* expression. In fact, it is impossible to define the $|\cdot|$ operator as a CODA function that maps an array of field elements to the integer that it represents (as a field element), as the array may represent an integer larger than the size of the finite field.

Specifying invariants. In order to prove that the CODA implementation of **BigLessThan** adheres to its specification, the developer also needs to annotate the **iter** construct with a so-called *loop invariant*, which specifies the refinement type of the accumulator. For this example, the annotation at lines 12-14 specifies that the accumulator is a pair of (binary) field elements, where the first element indicates whether the length-i prefix of **a** is less than the length-i prefix of **b**, and the second pair indicates whether they are equal. Because automated inference of loop invariants is, in general, not always possible, the CODA DSL requires the developer to annotate every usage of the **iter** construct with a refinement type specification of its accumulator and utilizes it in the verification process.

Verification. In order to prove that the implementation of **BigLessThan** conforms to its specification, CODA performs static type checking. In particular, CODA utilizes the refinement type annotations to generate a set of *proof obligations*, also referred to as *lemmas*. If all of these lemmas are proven to hold, this constitutes a formal proof that the implementation conforms to its specification. For our running example, CODA generates a total of 15 lemmas, one of which is the following:

$$\begin{aligned}
&\forall (k\ i : nat)(a\ b : F\ list),\\
&\quad length(a) = k \implies\\
&\quad length(b) = k \implies\\
&\quad 0 \le i < k \implies\\
&\quad |a[:i]| < |b[:i]| \implies\\
&\quad |a[:i+1]| < |b[:i+1]|.
\end{aligned}$$

Essentially, this lemma states that, if the length-i prefix of a represents an integer smaller than the length-i prefix of b, then the same is true for the length-$i+1$ prefixes. Note that this lemma is needed for proving the correctness of the loop invariant, which in turn is essential for establishing that the return value of **BigLessThan** has the intended refinement type specification.

Discharging proof obligations. Since proving properties about ZK applications often requires proving number-theoretic lemmas, there is unfortunately little hope of discharging the proof obligations generated by the CODA type checker *completely* automatically. Thus, rather than using an automated thorem prover (e.g., SMT solver), the CODA type checker instead outputs Coq theorems [23] to be proven interactively with the aid of CODA's domain-specific Coq tactics. In our running example, 10 of the lemmas can be discharged fully automatically and are not shown to the user. The user does need to be involved for proving the remaining 5 lemmas; however, CODA's tactics (as well as existing tactics provided by Coq and the Fiat crypto library [17]) are still useful for simplifying the proof.

Code generation. Once proof obligations are discharged and type checking succeeds, the CODA compiler generates a Rank-1 Constraint System (R1CS) [11] representation of the corresponding CODA program. We choose to compile CODA code down to R1CS because there are existing zkSNARK generators [22] that can be used to produce a prover and verifier from a set of R1CS constraints. For our running example, the R1CS constraints generated by the CODA compiler are roughly of the same size and quality as the constraints generated by the Circom compiler for the code from Figure 1.[2]

Threat model. We consider a trustless setup where the attacker has full access to the implementation of the ZK circuit, both at the source code level as well as the R1CS level. We also assume that the attacker has access to both the prover and the verifier generated from this circuit. We further assume that the attacker does not have to use the zero-knowledge proof generated by the prover; instead, they can replace the proof with their own bogus version.

3. The CODA Language

In this section, we present the syntax, semantics, types, and compilation procedure of the CODA DSL.

3.1. CODA Syntax and Semantics

At a high level, the CODA DSL is designed with three key goals in mind:

- **Programmability:** The DSL should make it possible to concisely and conveniently express arithmetic circuits of practical interest. To achieve this goal, CODA is modular and allows composing different circuits. It also incorporates higher-order combinators that allow mapping over arrays and performing aggregation.
- **Compilability:** Since our ultimate goal is to facilitate the generation of a ZK proof system (consisting of a prover and verifier), it should be possible to compile CODA programs down to R1CS constraints, for which existing techniques can generate ZK proof systems. To achieve this goal, CODA primarily allows computation over finite field elements and only allows using other values like integers and booleans in restricted ways.
- **Verifiability:** To allow CODA programs to be statically verified, CODA is both purely functional and based on a static refinement type system. Furthermore, to allow complex properties to be verified, CODA does not insist on decidable type checking. Instead, it outputs Coq lemmas

[1] While our refinement term language provides the $|x|$ operator as syntactic sugar, it can also be defined in terms of lower-level primitives in the type syntax.

[2] The size of the R1CS constraints is important both for efficiency and also for reducing the time and space costs of producing the proof [1].

r	$::=$	$\textbf{circuit}\ C\ \overrightarrow{(x_i : T_i)}\ \rightarrow T\ \{e\}$	**Circuit declaration**
e	$::=$		**Expression:**
	$\mid$	c	constants
	$\mid$	x	variable
	$\mid$	$\star$	wild card
	$\mid$	$b \mid z \mid f$	boolean/integer/field expression
	$\mid$	$\mathsf{assert}\ e_1 = e_2$	assertion
	$\mid$	$e_1 \oslash e_2$	binary relation
	$\mid$	$\#C\ \overrightarrow{e_i}$	circuit reference
	$\mid$	a	array operator
	$\mid$	t	product operator
	$\mid$	$\lambda x : \tau.\ e$	function abstraction
	$\mid$	$e_1\ e_2$	function application
	$\mid$	$\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$	variable binding
	$\mid$	$\mathsf{iter}_{\lambda i.\tau}\ e_s\ e_e\ e_f\ e_a$	iteration
b	$::=$	$\neg e \mid e_1 \odot e_2$	**Boolean expression**
z	$::=$	$-_{\mathbb{Z}} e \mid e_1 \oplus e_2$	**Integer expression**
f	$::=$	$-e \mid e_1 \otimes e_2$	**Field expression**
t	$::=$		**Product operator:**
	$\mid$	$(e_1, \ldots, e_n)_\phi$	product constructor
	$\mid$	$\mathsf{match}\, e\, \mathsf{with}\, (x_1, \ldots, x_n) \rightarrow e'$	product destructor
	$\mid$	$e.i$	product indexing
a	$::=$		**Array operator:**
	$\mid$	$e_1 :: e_2$	array constructor
	$\mid$	$\mathsf{map}\ e_1\ e_2$	map
	$\mid$	$\mathsf{length}\ e$	length
	$\mid$	$\mathsf{sum}\ e_1\ e_2\ e_3$	sum
	$\mid$	$e_1\,[e_2]$	array indexing
$\odot$	$\in$	$\{\wedge, \vee\}$	**Binary boolean operators**
$\oplus$	$\in$	$\{+_{\mathbb{Z}}, -_{\mathbb{Z}}, *_{\mathbb{Z}}\}$	**Binary integer operators**
$\otimes$	$\in$	$\{+, -, *, /\}$	**Binary field operators**
$\oslash$	$\in$	$\{=, <, \leq\}$	**Binary relations**

Figure 3. Syntax of CODA Programs

to be proven semi-automatically using a combination of proof tactics and manual intervention where necessary.

Figure 3 shows the syntax for CODA programs. Top-level CODA programs are referred to as *circuits* (parametrized over a finite field $\mathbb{F}$) and consist of three components: (1) $\overline{x_i : T_i}$, which is a sequence of input parameters x_i and their corresponding type T_i, (2) T, the output type of the circuit, and (3) e, the body of the circuit. Given some choice of values for $\overline{x_i}$, a CODA program evaluates the body expression e and returns the evaluation result v. At a high level, CODA is a functional programming language with domain specific constructs for building ZK circuits as well as several standard functional combinators. In what follows, we explain the key features of the CODA language.

Standard functional constructs in CODA include include integer and boolean expressions, arrays and tuples, lambda abstractions, function applications, iterators, and variable bindings. A variable binding $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ binds the result of evaluating expression e_1 to the (immutable) variable x and evaluates e_2. The expression $\mathsf{iter}_{\lambda i.\tau}\, e_s\ e_e\ e_f\ e_a$ performs aggregation over indices. Specifically, e_s, e_e are start and end indices respectively, e_f is a function, and e_a is the initial value of the accumulator. Starting with the initial value e_a, the function e_f is applied on the accumulator to produce a new value, and this process continues for all indices in the range $[e_s, e_e)$. Note that this construct has a subscript $\lambda i.\tau$ indicating the *loop invariant*, which is used in type-checking, as discussed in Section 3.3.

Beyond the standard functional constructs, CODA allows expressing computations over finite fields, which are crucial for building ZK circuits. Finite field expressions in CODA include field addition, subtraction, and multiplication. These operations are similar to their integer counterparts except that they occur modulo the field size $\mathbb{F}$ and may therefore over- or under-flow where their corresponding integer counterparts do not.

As CODA programs need to encode *relations* (rather than *mathematical functions*), they allow non-determinism, which is expressed via the $\star$ construct in the DSL, which can evaluate to *any* field element. Thus, a given CODA program can have multiple outputs even when executed on the same input. For example, a CODA function that takes as input a value x and "returns" $\star$ corresponds to a relation R that evaluates to true for any pair (x, y) (where y is the "output" of the CODA function).

In addition to allowing completely non-deterministic choices, CODA also allows constraining randomness via *assertions*. For example, consider the following CODA circuit:

```
circuit C (a : F) : F =
  let b = ⋆ in
  let _ = assert a * b = 0 in b
```

This circuit essentially corresponds to a relation $C(a, b)$ that evaluates to true if and only if $a * b = 0$ for some field elements a, b. Note that assertions are restricted to predicates of the form $e_1 = e_2$, as the resulting constraint must be compilable down to R1CS, which may not feasible for arbitrary predicates.

We conclude our discussion of CODA syntax with circuit references of the form $\#C(x)$. At a high-level, circuit references allow CODA programs to refer to other circuits in a *compositional* way. Operationally, they allow developers to pretend that CODA circuits can be called like regular functions (even though they are not). For instance, consider a CODA program that returns $\#C(x)$, where C is some circuit representing a relation $R(x, y)$. Then, the top-level CODA circuit C' can be viewed as syntactic sugar for the following program:

```
circuit C' (a : F) : F =
  let b = ⋆ in
  let _ = assert R(a, b) in b
```

3.2. CODA Semantics

The semantics of CODA require *valuations*, mappings from variables to values. We formalize the semantics of CODA expressions using judgments of the form $\sigma \vdash e \Downarrow v$, indicating that e can evaluate to value v under valuation σ. As emphasized earlier, CODA programs denote relations rather than mathematical functions; hence, it is possible for an expression e to evaluate to two distinct values v, v' under the same valuation.

While we present the semantics of the full CODA language in the appendix, Figure 4 shows the semantics for a representative subset of CODA expressions. The semantics utilize an environment Δ mapping circuits to their definitions in the E-CREF rule, and the iterator semantics are

v	$::=$		**Values:**
	$\mid$	c	constants
	$\mid$	$(v_1, \ldots, v_n)$	product value
	$\mid$	$v_1 :: v_2$	array value
	$\mid$	$\mathsf{Closure}(\lambda x : \tau.\ e,\ \sigma)$	closure

σ	$::=$		**Valuation:**
	$\mid$	$\varnothing$	empty valuation
	$\mid$	$\sigma[x \mapsto v]$	augmentation

$$\frac{}{\sigma \vdash v \Downarrow v}\ \textsc{E-Value} \qquad \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v}\ \textsc{E-Var} \qquad \frac{v \in \mathbb{F}}{\sigma \vdash \star \Downarrow v}\ \textsc{E-NonDet}$$

$$\frac{\sigma \vdash e_1 \Downarrow f \qquad \sigma \vdash e_2 \Downarrow f \qquad f \in \mathbb{F}}{\sigma \vdash \mathsf{assert}\ e_1 = e_2 \Downarrow \mathsf{unit}}\ \textsc{E-Assert} \qquad \frac{\sigma \vdash e_1 \Downarrow f_1 \qquad \sigma \vdash e_2 \Downarrow f_2 \qquad f_1 \otimes f_2 = f}{\sigma \vdash e_1 \otimes e_2 \Downarrow f}\ \textsc{E-FBinop}$$

$$\frac{}{\sigma \vdash \lambda x : \_.\ e \Downarrow \mathsf{Closure}(\lambda x : \_.\ e,\ \sigma)}\ \textsc{E-Lam} \qquad \frac{\sigma \vdash e_1 \Downarrow \mathsf{Closure}(\lambda x : \_.\ e,\ \sigma') \qquad \sigma \vdash e_2 \Downarrow v \qquad \sigma'[x \mapsto v] \vdash e \Downarrow v'}{\sigma \vdash e_1\ e_2 \Downarrow v'}\ \textsc{E-App}$$

$$\frac{\begin{array}{c}\sigma \vdash e_l \Downarrow v_l \quad \text{for all } l \in \{s, e, f, a\} \\ v_s \in \mathbb{Z} \qquad v_e \in \mathbb{Z} \qquad v_s \geq v_e\end{array}}{\sigma \vdash \mathsf{iter}_{\lambda i.\tau}\ e_s\ e_e\ e_f\ e_a \Downarrow v_a}\ \textsc{E-Iter0} \qquad \frac{\begin{array}{c}\sigma \vdash e_l \Downarrow v_l \quad \text{for all } l \in \{s, e, f, a\} \\ v_s \in \mathbb{Z} \qquad v_e \in \mathbb{Z} \qquad v_s < v_e \qquad \sigma \vdash \mathsf{iter}_{\lambda i.\tau}\ (v_s +_{\mathbb{Z}} 1)\ v_e\ v_f\ (v_f\ v_a) \Downarrow v\end{array}}{\sigma \vdash \mathsf{iter}_{\lambda i.\tau}\ e_s\ e_e\ e_f\ e_a \Downarrow v}\ \textsc{E-IterS}$$

$$\frac{\Delta(C) = \textbf{circuit}\ \ R\ (x_1 : T_1) \cdots (x_n : T_n)\ \ \to T\ \{e\} \qquad \sigma \vdash e_i \Downarrow v_i \quad \text{for all } i \in \{1, \ldots, n\} \qquad \sigma[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] \vdash e \Downarrow v}{\sigma \vdash \#C\ e_1 \cdots e_n \Downarrow v}\ \textsc{E-CRef}$$

Figure 4. Selected Semantics Rules of CODA Expressions

T	$::=$		**Basic types:**
		F	field element
	$\mid$	Int	integer
	$\mid$	Bool	boolean
	$\mid$	$T_1 \times \cdots \times T_n$	product (Unit denotes empty)
	$\mid$	$[T]$	array type
	$\mid$	$\{\nu : T \mid \phi\}$	refinement type
τ	$::=$	T	
	$\mid$	$x : \tau_1 \to \tau_2$	Function type
ϕ	$::=$		**Logical qualifiers:**
	$\mid$	t	refinement term
	$\mid$	$\neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$	boolean operators
	$\mid$	$\forall_{i \in [0,k)}.\ \phi$	bounded quantification
t	$::=$		**Refinement terms:**
		$\mathsf{true} \mid \mathsf{false}$	boolean constants
	$\mid$	F	field size
	$\mid$	e	expression
	$\mid$	$\mathsf{toN}(t) \mid \mathsf{toZ}(t)$	field to integer conversion

Figure 5. Syntax of CODA Types

defined recursively. Finally, note that, for assert statements, evaluation gets stuck for field elements that do not satisfy the predicate. Thus, if we think of a CODA circuit C with inputs $\overline{x}$ and body e as a relation $R_C((x_1, \ldots, x_n), y)$, we have $R_C((in_1, \ldots, in_n), out)$ evaluates to true if and only if $(x_1 \mapsto in_1, \ldots, x_n \mapsto in_n) \vdash e \Downarrow out$.

3.3. CODA Type System

CODA facilitates static verification through its rich type system based on *refinement types*. At a high level, refinement types allow developers to express functional correctness requirements on their ZK circuits as type annotations that must be checked by the compiler.

Figure 5 shows the syntax for CODA's refinement type system, which can be split into four core classes, namely, *base types*, *compound types*, *function types*, and *refinement types*. The three base types include F, Int, and Bool, indicating field elements, integers, and booleans respectively. As standard, compound types include array types $[T]$, denoting an array of elements of type T, and product types, $T_1 \times \ldots \times T_n$, denoting tuples $(e_1, \ldots, e_n)$ where each tuple element e_i has type T_i. Function types are of the form $x : \tau_1 \to \tau_2$, where τ_1 is the type of the input and τ_2 is the type of the output. The crucial part of the CODA type system are refinement types of the form $\{\nu : T \mid \phi\}$ which qualify standard types with an additional *logical qualifier* ϕ that the value must satisfy. As a simple example, the refinement type $\{\nu : \mathsf{F} \times \mathsf{F} \mid \nu.1 = \nu.2\}$ describes a pair whose first element is equal to its second element. In more detail, logical qualifiers in CODA are (possibly universally quantified) boolean combinations of atomic predicates involving terms t. Terms in the refinement type syntax include all expressions allowed in the CODA syntax as well as built-in functions like toN and toZ used for converting fields to signed and unsigned integers, respectively. We found such field-element-to-integer conversion functions to be crucial for concisely expressing many specifications of practical interest; hence, we include them as primitives in our type syntax. However, we also note that CODA's refinement term language is *extensible*, meaning that the developer can define additional functions to use as part of their specification.

3.3.1. Type Checking Rules.

While refinement types allow developers to write specifications for their ZK circuits, we still need an algorithm that can be used to *statically verify* the type annotations, which is the job of the type checker. We formalize CODA's type checking algorithm using three

$$\frac{\Delta; x_1 : T_1, \ldots, x_n : T_n \vdash e : T_y}{\Delta \vdash_C \textbf{circuit}\ \ C\ (x_1 : T_1) \cdots (x_n : T_n)\ \ \to T_y\ \{e\}}\ \textsc{T-CircDecl}$$

$$\frac{\Gamma(x) = \{\nu : T \mid \phi\}}{\Delta; \Gamma \vdash x : \{\nu : T \mid \nu = x\}}\ \textsc{TE-Var} \qquad \frac{\Gamma(f) = x : \tau_1 \to \tau_2}{\Delta; \Gamma \vdash f : (x : \tau_1 \to \tau_2)}\ \textsc{TE-Var-Func}$$

$$\frac{}{\Delta; \Gamma \vdash \star : \{\nu : \mathsf{F} \mid \mathsf{true}\}}\ \textsc{TE-NonDet} \qquad \frac{f \in \mathbb{F}_p}{\Delta; \Gamma \vdash f : \{\nu : \mathsf{F} \mid \nu = f\}}\ \textsc{TE-ConstF}$$

$$\frac{\Delta; \Gamma \vdash x : \{\nu : \mathsf{F} \mid \phi_1\} \qquad \Delta; \Gamma \vdash y : \{\nu : \mathsf{F} \mid \phi_2\}}{\Delta; \Gamma \vdash \mathsf{assert}\ x = y : \{\nu : \mathsf{Unit} \mid x = y\}}\ \textsc{TE-Assert} \qquad \frac{\Delta; \Gamma \vdash x : \{\nu : \mathsf{F} \mid \phi_1\} \qquad \Delta; \Gamma \vdash y : \{\nu : \mathsf{F} \mid \phi_2\}}{\Delta; \Gamma \vdash x \otimes y : \{\nu : \mathsf{F} \mid \nu = x \otimes y\}}\ \textsc{TE-BinopField}$$

$$\frac{\Delta; x : \tau_1, \Gamma \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.\ e : (x : \tau_1 \to \tau_2)}\ \textsc{TE-Abs} \qquad \frac{\Delta; \Gamma \vdash x_1 : (s : \tau_s \to \tau_r) \qquad \Delta; \Gamma \vdash x_2 : \tau_s}{\Delta; \Gamma \vdash x_1\ x_2 : \tau_r[s \mapsto x_2]}\ \textsc{TE-App}$$

$$\frac{\begin{array}{c} s, e, j, f, a \text{ fresh} \qquad \tau_f = j : \{\nu : \mathsf{Int} \mid s \le \nu < e\} \to \tau[i \mapsto j] \to \tau[i \mapsto j+1] \\ \tau_a = \tau[i \mapsto s] \qquad \tau_r = \tau[i \mapsto e] \end{array}}{\Delta; \Gamma \vdash \mathsf{iter}_{\lambda i.\tau} : s : \mathsf{Int} \to e : \mathsf{Int} \to f : \tau_f \to a : \tau_a \to \tau_r}\ \textsc{TE-Iter}$$

$$\frac{\Delta(C) = \textbf{circuit}\ (x_1 : T_1) \cdots (x_n : T_n)\ \ \to (y : T_y)}{\Delta; \Gamma \vdash \#C\ \ : (x_1 : T_1 \to \cdots \to x_n : T_n \to T_y)}\ \textsc{TE-CRef} \qquad \frac{\Delta; \Gamma \vdash e : \tau' \qquad \Gamma \models \tau' <: \tau}{\Delta; \Gamma \vdash e : \tau}\ \textsc{TE-Sub}$$

Figure 6. Selected Typing Rules for CODA expressions and programs

kinds of typing judgments, namely (1) a circuit typing judgment, (2) an expression typing judgment, and (3) a subtyping judgment, which utilize the following environments:

- **Circuit store**: The *circuit store* Δ keeps track of previously defined circuits. In particular, it maps the name of a circuit to the input and output type and is used for type checking circuit references.
- **Type environment:** The *type environment* Γ maps variable names to their types and is used to type check variable references.

Figure 6 presents our system's typing rules. We elided some of the standard, uninteresting rules for space.

Circuit typing. The top-level typing judgment, labeled T-CIRCDECL, is used to type check CODA circuits. If the body e of the circuit has type T_y under the assumption that each argument x_i has type T_i, then the whole circuit is well-typed.

Expression typing. We describe well-typed expressions with the typing judgment $\Delta; \Gamma \vdash e : \tau$, meaning that "under context Γ, the expression e has type τ." Due to space constraints, we only describe the most important expression typing rules in the following discussion; however, the interested reader can find all expression typing rules (together with their soundness proof) in the Appendix. We start with the standard rules first and work our way down to the more involved type checking rules.

Variables. The rules TE-VAR and TE-VAR-FUNC are used for typing variables. In both of these rules, the type of the variable is found in the context. For non-functional variables, we additionally retain the information that the returned data is the same as what is present in the variable itself. This enables simple, lightweight equality reasoning.

Constants. The rule TE-NONDET and TE-CONSTLIST are used for typing constants. Note that $\star$ corresponds to non-deterministically chosen field elements; hence, $\star$ always has type F. Furthermore, a field constant f has type $\{\nu : \mathsf{F} \mid \nu = f\}$, which essentially corresponds to the singleton set $\{f\}$. (The typing rules for boolean and integer constants are similar.)

Assertions. The type checking rule for assertions, labeled TE-ASSERT, keeps track of the constraint being asserted in the type environment. Assertions only ensure the asserted information so the base type is Unit.

Binops. The rule TE-BINOPFIELD deals with binary expressions. For a given expression $x \otimes y$ to type check, x and y must both be field elements. The type of the expression is then $\{\nu : T \mid \nu = x\ \otimes\ y\}$. The elided binop rules, defined on data types like booleans, follow a similar structure.

Functions. The rules for function abstraction (TE-ABS) and application (TE-APP) are standard: The abstraction rule binds the type of x to its declared type τ_1 and checks that e_2 has type τ_2 under this assumption. The application rule $x_1 x_2$ checks that x_1 has function type $s : \tau_s \to \tau_r$ and that x_2 has type τ_s. Then, the resulting type of the application is the return type τ_r, with occurrences of s in τ_r substituted by x_2.

Iteration. The type checking rule for the iter construct essentially checks the validity of the annotated loop invariant. In particular, recall that the iter construct is annotated with an invariant of the form $\lambda i.\tau$, indicating that the result of the aggregation has type τ for each index i. Hence, this rule asserts that (1) the initial value of the accumulator has type $\tau[i \mapsto s]$ (where s is the start index), (2) the return value has type $\tau[i \mapsto e]$ (where e is the end index), and (3) the function f has a suitable type of the form $\tau_f \equiv j : \tau_1 \to (\tau_2 \to \tau_3)$, where τ_1 states that the index j is in the range $[s, e)$ and the return type $\tau_2 \to \tau_3$ states that the accumulator invariant τ is preserved, since $\tau_2 \equiv \tau[i \mapsto j]$ and $\tau_3 \equiv \tau[i \mapsto j+1]$.

$$\frac{}{\Gamma \vDash \tau <: \tau}\ \textsc{TSub-Refl}$$

$$\frac{\Gamma \vDash \tau_1 <: \tau_2 \qquad \Gamma \vDash \tau_2 <: \tau_3}{\Gamma \vDash \tau_1 <: \tau_3}\ \textsc{TSub-Trans}$$

$$\frac{\begin{array}{l}\Gamma \vDash T_1 <: T_2 \qquad P \equiv \llbracket \phi_1 \rrbracket \implies \llbracket \phi_2 \rrbracket \\ \forall \vec{x} \in \mathsf{dom}(\Gamma).\ \forall \nu.\ \mathsf{Encode}(\Gamma) \implies P\end{array}}{\Gamma \vDash \{\nu : T_1 \mid \phi_1\} <: \{\nu : T_2 \mid \phi_2\}}\ \textsc{TSub-Refine}$$

$$\frac{\begin{array}{c}\Gamma \vDash \tau_y <: \tau_x \\ z \text{ fresh} \qquad \Gamma \vDash \tau_r[x \mapsto z] <: \tau_s[y \mapsto z]\end{array}}{\Gamma \vDash (x : \tau_x \to \tau_r) <: (y : \tau_y \to \tau_s)}\ \textsc{TSub-Fun}$$

$$\frac{\Gamma \vDash T_1 <: T_2}{\Gamma \vDash [T_1] <: [T_2]}\ \textsc{TSub-Array}$$

$$\frac{\Gamma \vDash T_i <: T_i' \quad \text{for all } i \in \{0, \ldots, n\}}{\Gamma \vDash T_1 \times \ldots \times T_n <: T_1' \times \ldots \times T_n'}\ \textsc{TSub-Product}$$

Figure 7. Subtyping rules for CODA

Circuit references. Circuit references are type-checked similarly to function applications. In particular, if C is a circuit with n inputs of type $T_1, \ldots, T_n$ and an output of type τ_y, then the expression $\#C$ has type $x_1 : T_1 \to \ldots \to x_n : T_n \to T_y$.

3.3.2. Subtyping. The type checking rules described in the previous section make use of a *subtyping* relation: Specifically, the TE-SUB rule from Figure‘6 states that an expression of type τ' also has type τ as long as τ' is a subtype of τ, denoted as $\Gamma \vDash \tau' <: \tau$. In this subsection, we discuss the subtyping relation, presented in Figure 7, underlying CODA's type system.

The first two rules, TSUB-REFL and TSUB-TRANS, ensure that the subtyping relation is reflexive and transitive.

The third rule, called TSUB-REFINE, is the crux of the entire type system and reduces refinement type checking to querying logical validity in some formal logic. In particular, a refinement type $\{\nu : T_1 \mid \phi_1\}$ is considered a sub-type of $\{\nu : T_2 \mid \phi_2\}$ if the following two conditions hold: First, the base type T_1 must be a subtype of T_2. Second, the logical qualifier ϕ_2 must be logically entailed by ϕ_1 under the type environment Γ. To perform the second check, the CODA type system generates a logical encoding of ϕ_1, ϕ_2 and Γ in Coq and produces a lemma that must be proven in order for type checking to succeed. We discuss the process of discharging proof obligations in Coq in the next subsection.

The next rule called TSUB-FUN is used for deciding sub-typing between functions. As standard, this rule states that subtyping for functions is contravariant with respect to the argument types and covariant with respect to the return types. The next two rules for arrays and products are also standard and state that subtyping is covariant with respect to the nested element types.

3.4. Discharging Proof Obligations in Coq

Because logical qualifiers in the CODA type system make number-theoretic statements about finite field elements, the lemmas produced by the CODA type checker cannot by discharged automatically in general. Hence, we adopt a *semi-automated* approach, based on Coq, for discharging the proof obligations generated by the CODA type checker. We choose to use Coq for this purpose because of the existence of libraries for finite field arithmetic as well as a rich dictionary of tactics for proof automation.

Recall from the previous subsection that subtyping checks in the CODA type system may involve an application of TSUB-REFINE rule, which has the following premise:

$$\forall \vec{x} \in \mathsf{dom}(\Gamma).\ \forall \nu.\ \mathsf{Encode}(\Gamma) \implies \llbracket \phi_1 \rrbracket \implies \llbracket \phi_2 \rrbracket$$

This logical validity query is translated into a Coq proposition by translating the formulas ϕ_1, ϕ_2 as well as the typing environment into Coq objects. Hence, each application of the TSUB-REFINE rule during type checking corresponds to a Coq proposition that must be discharged, so type checking is only valid under the assumption that the resulting proof obligations have been discharged. As an example, Figure 8 shows one of the proof obligations generated for the **BigLessThan** example from Section 2. This Coq proposition essentially states that the output signal, *out*, of the circuit actually indicates whether $|xs| < |ys|$ under the assumption that the **iter** invariant holds.

$$\begin{array}{l}\forall (k : nat)(xs\ ys : list\ F)(out\ lt : F), \\ \ 2 \leq k \implies \\ \ length(xs) = length(ys) = k \implies \\ \ binary(xs) \wedge binary(ys) \implies \\ \ lt = |xs[:k]| <?\ |ys[:k]| \implies \\ \ out = lt \implies \\ \ out = |xs| <?\ |ys|.\end{array}$$

Figure 8. A proof obligation for the **BigLessThan** program

Discharging a proof obligation requires providing a formal Coq proof of the corresponding proposition. As stated earlier, Coq has a number of proof tactics that can be used to automate proofs, so some of the simpler proof obligations can be discharged automatically or with very little manual effort, particularly with the help of the Fiat-Crypto library [17]. For instance, 10 proof obligations for the **BigLessThan** example can be discharged in a fully automated way via Coq's standard **auto** tactic, and an additional 3 may be discharged trivially using fewer than four standard tactics.

In addition to the standard tactics provided by Coq and the Fiat-Crypto library [17] (which we use for representing finite fields), we identified a number of other opportunities for automatically discharging common proof obligations generated by the CODA type checker. Table 1 summarizes some of the tactics that we found to be useful when proving CODA-generated Coq propositions. For example, because statements about integers operations must be implemented using field operations, which can overflow, we provide tactics that can automatically prove the field operations are

Tactic	Summary
F_to_Z	Rewrite a field expression into an integer expression, generating overflow-safety side-conditions
overflow	Solve overflow-safety obligations
split_sum	Split a summation into disjoint sub-summations
switch_sum	Exchanges two nested summations
reduce_sum	Reduce trivial summations to
ind	Perform case analysis on an binary field element

TABLE 1. SELECTED CODA COQ TACTICS.

overflow-safe. First, the **F_to_Z** tactic rewrites a field expression into an equivalent integer expression, generating integer inequality side-conditions that ensure overflow safety. However, the generated side-conditions often involve nonlinear arithmetic, such as multiplication and exponentiation, making automated solving undecidable. To mitigate this problem, we provide a domain-specific **overflow** tactic that performs backtracking search and attempts to reduce nonlinear inequalities into equivalent linear inequalities based the user-provided bounds on existing variables. Finally, the resulting linear inequalities are solved by existing decision procedures such as Coq's **lia** tactic. As another example, because ZKP programs build functionalities using field *arithmetics* such as addition and multiplication, the proofs often need to reason about indexed summations. To this end, we provide the **split_sum** tactic that splits a summation into multiple disjoint summations, **switch_sum** that exchanges two nested summations, and **reduce_sum** to reduce trivial summations into simpler expressions.

3.5. Properties of Well-Typed Coda Programs

In this section, we present the key correctness guarantees established by well-typed CODA programs and refer the reader to section B for the proofs. The correctness of CODA's type system is characterized by expression type preservation. The preservation theorem states that if e is a well-typed expression with type τ and e evaluates to some value v, then v also has the type τ.

Type preservation requires one additional assumption – well-typed valuations. We describe well-typed valuations with the judgment $\Delta; \Gamma \vDash \sigma$, meaning that "under context Γ, the values of all variables in σ have the same types as those in Γ." The interested reader can find the formal definition of valuation typing in the Appendix. This assumption ensures that the arguments provided to a circuit are well-typed, for if the arguments to the circuit are not well-typed then the resulting value may also not be well-typed.

Theorem 1 (Expression Type Preservation)**.** *If* $\Delta; \Gamma \vdash e : \tau$ *and* $\sigma \vdash e \Downarrow v$ *and* $\Delta; \Gamma \vDash \sigma$*, then* $\Delta; \Gamma \vdash v : \tau$*.*

As a corollary, Theorem 1 immediately implies the following: If (1) C is a well-typed CODA program and $\vec{e}$ are input values satisfying the assumptions specified in the input types of C, and (2) $\#C\ \vec{e}$ evaluates to output values $\vec{v}$, then (3) the output values $\vec{v}$ are guaranteed to satisfy the output types of C, as stated in the following theorem:

Theorem 2 (Circuit Evaluation Type Preservation)**.** *If* $\Delta; \Gamma \vdash \#C\ \vec{e} : \tau$ *and* $\sigma \vdash \#C\ \vec{e} \Downarrow v$ *and* $\Gamma \vDash \sigma$*, then* $\Delta; \Gamma \vdash v : \tau$*.*

4. The CODA Compiler

In this section, we describe the CODA compiler for converting CODA programs to R1CS constraints. Because zero-knowledge proof systems (based on zkSNARKs [9]) generate the prover and verifier from a Rank 1 Constraint system, the CODA compiler needs to generate R1CS in order to leverage existing frameworks for zkSNARK generation. More formally, a Rank-1 constraint system (V, Φ) consists of a set of variables V and a list Φ of second-degree polynomial constraints of the form $A * B + C = 0$, where A, B and C are field constants or variables in V. The goal of the CODA compiler is to translate CODA programs to a set of R1CS constraints of this form.

We describe our compilation technique in Figure 4 using judgments of the following shape:

$$\sigma \vdash e \rightsquigarrow \langle \Phi, u \rangle$$

indicating that, under (symbolic) valuation σ, expression e evaluates to a *compilation value* u, with a corresponding set Φ of R1CS constraints. A compilation value u is essentially a partially evaluated version of expression e, where the original expression is simplified as much as possible and brought into an irreducible form. Because many CODA expressions are not valid R1CS terms, the primary goal of partial evaluation is to facilitate the generation of valid R1CS constraints.

Figure 4 shows a representative subset of the compilation rules underlying the CODA compiler. The first rule called C-NONDET simply introduces a fresh R1CS variable r representing a non-deterministically chosen field element. The next rule called C-ASSERT adds a new constraint $u_1 = u_2$ where u_1 and u_2 are the results of partially evaluating expressions e_1 and e_2 respectively. The next two rules with prefix C-BINOP compile binary field expressions of the form $e_1 \otimes e_2$. In the reducible case, both e_1 and e_2 are partially evaluated as constansts ($u_1, u_2 \in \mathbb{F}$), so we further reduce $e_1 \otimes e_2$ by evaluating $u_1 \otimes u_2$. Otherwise, in the irreducible case, the compilation result is the symbolic expression $u_1 \otimes u_2$. [3]

The final rule called C-CIRC in Figure 4 presents the compilation procedure for the entire circuit. Given a circuit C with parameters $x_1, \dots, x_n$ and body e, the compilation procedure introduces fresh symbolic values r_i for each parameter x_i and then compiles e into (Φ, u). The final R1CS encoding of the circuit is then obtained as $\Phi \cup r_{n+1} = u$, where the fresh variable r_{n+1} corresponds to the circuit output.

[3] The careful reader may notice that a generated constraint can be made between irreducible field expressions whose degrees may exceed 2, which violates the "rank-1"-ness of R1CS. However, this is non-problematic, as we can introduce fresh R1CS variables to represent sub-expressions and reduce the degree to 2. For example, the constraint $x = y * y * y$ can be transformed into the equivalent constraints $z = y * y, x = y * x$ where z is fresh.

$$
\begin{array}{llll}
u & ::= & & \textbf{Compilation values:} \\
 & \mid & c & \text{constants} \\
 & \mid & (u_1, \ldots, u_n) & \text{product value} \\
 & \mid & u_1 :: u_2 & \text{array value} \\
 & \mid & \mathsf{Closure}(\lambda x : \tau.\ e,\ \sigma) & \text{closure} \\
 & \mid & r & \text{R1CS variable} \\
 & \mid & r \otimes u \mid u \otimes r & \text{Irreducible expressions} \\
\sigma & ::= & & \textbf{Valuation:} \\
 & \mid & \varnothing & \text{empty valuation} \\
 & \mid & \sigma[x \mapsto u] & \text{augmentation}
\end{array}
$$

$$
\frac{}{\sigma \vdash u \rightsquigarrow \langle \varnothing, u \rangle}\ \textsc{C-Value}
\qquad
\frac{r \text{ fresh}}{\sigma \vdash \star \rightsquigarrow \langle \varnothing, r \rangle}\ \textsc{C-NonDet}
$$

$$
\frac{\begin{array}{c} \sigma \vdash e_1 \rightsquigarrow \langle \Phi_1, u_1 \rangle \\ \sigma \vdash e_2 \rightsquigarrow \langle \Phi_2, u_2 \rangle \end{array}}{\sigma \vdash \mathsf{assert}\ e_1 = e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2 \cup \{u_1 = u_2\}, \mathsf{unit} \rangle}\ \textsc{C-Assert}
$$

$$
\frac{\begin{array}{c} \sigma \vdash e_1 \rightsquigarrow \langle \Phi_1, u_1 \rangle \qquad \sigma \vdash e_2 \rightsquigarrow \langle \Phi_2, u_2 \rangle \\ u_1, u_2 \in \mathbb{F} \\ u_1 \otimes u_2 = u_3 \end{array}}{\sigma \vdash e_1 \otimes e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2, u_3 \rangle}\ \textsc{C-BinopReducible}
$$

$$
\frac{\begin{array}{c} \sigma \vdash e_1 \rightsquigarrow \langle \Phi_1, u_1 \rangle \qquad \sigma \vdash e_2 \rightsquigarrow \langle \Phi_2, u_2 \rangle \\ u_1 \notin \mathbb{F} \text{ or } u_2 \notin \mathbb{F} \end{array}}{\sigma \vdash e_1 \otimes e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2, u_1 \otimes u_2 \rangle}\ \textsc{C-BinopIrreducible}
$$

$$
\frac{\begin{array}{c} r_i \text{ fresh for all } i \in \{1, \ldots, n+1\} \\ [x_1 \mapsto r_1] \cdots [x_n \mapsto r_n] \vdash e \rightsquigarrow \langle \Phi, u \rangle \end{array}}{\textbf{circuit}\ C\ (x_1 : \mathsf{F}) \cdots (x_n : \mathsf{F})\ \to \mathsf{F}\ \{e\} \rightsquigarrow_C \Phi \cup \{r_{n+1} = u\}}\ \textsc{C-Circ}
$$

Figure 9. Rules for compiling DSL program to R1CS constraints

Due to lack of space, Figure 4 does not show the compilation rules for *all* CODA expressions, and the remaining compilation rules are basically the same as the operational semantics. Instead of formally describing the remaining compilation rules in detail, we remark on a few salient features of the remaining rules. First, because circuit inputs and the $\star$ expression in CODA correspond to field elements, all boolean and integer expressions can be compiled down to constants. As a corollary, CODA can unroll all loops during compilation. Furthermore, because array lengths in CODA are fixed, the CODA compiler can statically infer the exact shape of arrays. Hence, array operations, such as indexing and mapping, can also be fully evaluated.

5. Implementation

In this section, we provide more details about the implementation of CODA, which is largely implemented in OCaml and consists of around 4000 lines of source code.

The CODA language We implemented CODA as a embedded domain-specific language (EDSL) in OCaml. That is, CODA programs are expressed as OCaml expressions built from a library of atomic building blocks that represent CODA constructs. The choice of implementing CODA as an EDSL means that the developer is allowed to use features from the host language (e.g., OCaml's type system, module system, as well as the expansive standard and third-party libraries) to significantly bootstrap the development speed.

Type Checking The type checker implements the type checking and subtyping rules shown in Figures 6 and 7. Before type checking, CODA programs are normalized to the administrative normal form (ANF), and types are normalized to disallow consecutive refinements (i.e. refinement types of the form $\{\nu : \{\nu : T \mid \phi_1\} \mid \phi_2\}$ are normalized to $\{\nu : T \mid \phi_1 \wedge \phi_2\}$).

Lemma generation Recall that the TSUB-REFINE subtyping rule proves that one refinement type is the subtype of another by requiring that implication of their qualifiers is a valid Coq proposition. Thus, the CODA type checker collects all such implications generated during type checking, and translates them into equivalent Coq lemmas. The translation is mostly standard: a base type is embedded as the corresponding Coq type, an array type is embedded into Coq's **list** type refined with its length, a product type is embedded as Coq's product type, and a refinement type $\{\nu : T \mid \phi\}$ is embedded as T's embedding with a hypothesis that asserts the truth of the embedding of ϕ. We also define interpretations of CODA operators that may appear in ϕ. Finally, we use Fiat-Crypto [17] for the formalization of finite fields.

Extensibility of CODA Type System In case the user has difficulty encoding a desired property P using CODA's refinement type (e.g., if P involves unbounded quantification), we allow the user to extend the type system by treating P as an opaque, uninterpreted predicate. The CODA type checker will run as normal, but it may generate proof obligations containing P. Thus, the user must also provide the interpretation of P in the form of a Coq definition.

Code generation The CODA compiler implements rules shown in Figure 4 to partially evaluate CODA programs into constraints over field expressions. We highlight some details that are elided in the compilation rules. First, the compiler relaxes the restriction on the circuit's input type by allowing non-field inputs, as long as their concrete valuation has been provided. During compilation, the compiler resolves any array length into a concrete value such that array operations can be fully evaluated. Finally, the collected constraints undergo a global transformation pass that reduces the degree of each constraint to at most 2, and eliminates any redundant constraints. Finally, the CODA compiler outputs an R1CS file in the zkInterface binary format for interoperating with different zk-SNARK backends.

Verified library Thanks to the functional aspect of CODA, the core CODA constructs can be easily composed to write expressive, higher-order *library* functions that can be reused across applications. Those functions can be verified just like circuits by checking the implementation against the specified types and proving the resulting lemmas in Coq. We have implemented this idea by building a verified library that encapsulates common ZKP patterns such as branching, functional zip, element-wise array arithmetics and array aggregate operations. Applications that use those library functions not only benefit from shorter, easier to understand programs, but also experience less proof burden, as those library functions have been fully certified.

6. Evaluation

In this section, we describe the results of an evaluation that is designed the answer the following research questions.

- **RQ1**: Can we use CODA to build formally verified implementations of commonly-used ZK circuits?
- **RQ2**: Can CODA help discover correctness bugs?
- **RQ3**: Can CODA proof automatic tatics help reduce the manual verification effort?
- **RQ4**: Can the CODA compiler generate efficient R1CS code and how does it compare against Circom?

Benchmarks. To answer these questions, we collected 79 ZK circuits sampled from 9 widely-used libraries and projects implemented in Circom. The libraries include `circomlib`, the standard Circom library, `circom-bigint`, a popular big-integer library for Circom, and the top-7 GitHub repositories under the "circom" topic ranked by the number of GitHub stars.

For each library, we selected the circuits used in our evaluation based on the following criteria: 1) **Documentation:** The original Circom code should have sufficient documentation to allow us to formally specify its behavior. 2) **External dependencies:** If the program has an external dependency, we require the external dependency to also satisfy the first criterion.

Experiment Setup. To perform our experiments, we manually translated these benchmarks to CODA in a way that preserves their semantics. We expressed the functional correctness requirements for each circuit as refinement type annotations and added invariant annotations wherever required in the CODA program. All experiments reported in this section are conducted on a MacBook Pro with Apple M1 Pro CPU and 16 GB of memory. CODA uses OCaml version 4.13.1 and Coq version 8.15.2. We use Circom 2.1.2 to compare with our generated R1CS constraints.

6.1. Main Results

Table 2 summarizes the results of certifying our benchmarks using CODA. Here, the column called "Original LOC" shows to the lines of code of the original Circom circuit. In the "CODA Program" group, the columns "LOC", "Spec" and "LOC/Spec" show the size of translated program in terms of lines of code, the lines of specification (including the refinement type and loop invariants, if any), and the ratio of program size to specification size. Next, the "Proof Obligations" group displays statistics about the proof obligations generated by the CODA type checker. In particular, columns "# (avg)", "# (total)" and "# Provable" shows the average number of proof obligations per circuit, the total number of obligations, and the number of provable obligations respectively. The column called "Discharged" shows the percentage of correct lemmas that can be discharged (either automatically or interactively), and "Auto" shows the percentage of those that can be discharged fully automatically. Finally, the "Avg Proof Length" column shows the average number of lines for the Coq proof, and the "Avg TC Time" column shows the average running time of the type checker in milliseconds.

Our key result is that, for 84% of the Circom programs that are originally correct, we were able to prove the correctness of our CODA translation. Furthermore, the CODA implementation of these circuits are on average 83% shorter than the original Circom implementation, and the specifications require 16 lines of type annotations on average. Overall, the specification-to-implementation ratio for CODA program is 0.6, suggesting that CODA does not impose an undue burden on programmers for writing specifications. In terms of the burden of proving these specifications, the CODA type checker outputs an average of 5 proof obligations as Coq lemmas, of which 76% discharged fully automatically. For the remaining ones, the average proof length (using our tactic library) is 10 lines of Coq code.

Result for RQ1: CODA is able to certify 84% of the non-buggy benchmarks, achieving a program–to–specification ratio of 0.6 and requiring only 10 lines of manual proofs.

6.2. Vulnerability Discovery

Because CODA is sound, a buggy program will lead to erroneous lemmas that will not be provable in Coq. When performing this evaluation, we discovered 6 benchmarks that resulted in proof obligations we were unable to discharge. Upon further inspection, we discovered that these failed proofs were due to subtle (and previously unknown) correctness bugs in the original Circom circuits.

Table 3 summarizes the vulnerabilities uncovered during our evaluation. In this table, the "Library" and "Program" columns show the library and program in which the vulnerability resides, respectively. The "Bug" column summarizes the root cause of the vulnerability, while "Patch Pull Request" links to the pull request that we created that would render the program certifiable by CODA. Note that all of the uncovered bugs are due to important logical constraints that are missing in the original program. Hence, these vulnerabilities can be exploited by a malicious prover to produce bogus zero-knowledge proofs for untrue statements. We have informed the developers of all of the affected libraries of the vulnerabilities and their corresponding fix.

Result for RQ2: CODA helped reveal 6 previously unknown vulnerabilities in widely-used ZK circuits.

6.3. Case Study: Reduction in Verification Effort

In this section, we investigate CODA's effectiveness in reducing the verification effort. Specifically, we compare the effort of certifying ZKP programs using CODA versus directly certifying the original Circom programs in Coq. In the latter approach, the source programs are syntactically translated into functions in Coq, a technique known as *shallow embedding*. For example, using shallow embedding, a Circom variable is encoded into a Coq variable, and

Library	Original	CODA Program			Proof Obligations					Proof Length	TC Time
	LOC	LOC	Spec	Spec/LOC	# Avg	# total	Provable	Discharged	Auto	(avg lines)	(avg ms)
Circomlib	114	37	21	0.57	3	42	42	100%	76%	6	177
Circom-bigint	264	74	38	0.51	9	123	123	81%	69%	33	182
Semaphore	71	23	22	0.96	4	22	22	100%	91%	6	181
Sismo	121	60	35	0.58	8	32	32	100%	94%	3	184
ZK–SBT	88	29	18	0.62	6	37	37	100%	76%	4	181
Darkforest–eth	29	10	5	0.50	4	13	13	100%	77%	5	180
ZK–SQL	47	12	4	0.33	3	10	10	70%	80%	4	180
Circomlib–ml	27	13	4	0.31	3	10	10	83%	70%	4	181
ed25519–circom	50	27	16	0.59	3	17	9	100%	67%	6	182

TABLE 2. MAIN RESULTS

Library	Program	Bug
Circom–bigint	BigMod	Missing range check on remainder
Circomlib–ml	IsPositive	Zero regarded as positive
ed25519–circom	LessThanPower	Missing logical constraints
ed25519–circom	LessThanBounded	Missing logical constraints
ed25519–circom	fulladder	Missing logical constraints
ed25519–circom	onlycarry	Missing logical constraints

TABLE 3. ZERO–DAY VULNERABILITIES DISCOVERED IN THE BENCHMARKS.

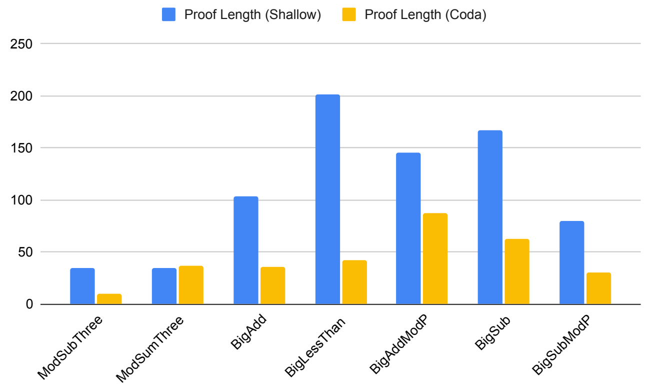

Figure 10. Comparison of verification effort of certifying the Circom-bigint library using CODA vs. using shallow embedding. The vertical axis shows the average number of constraints in log scale.

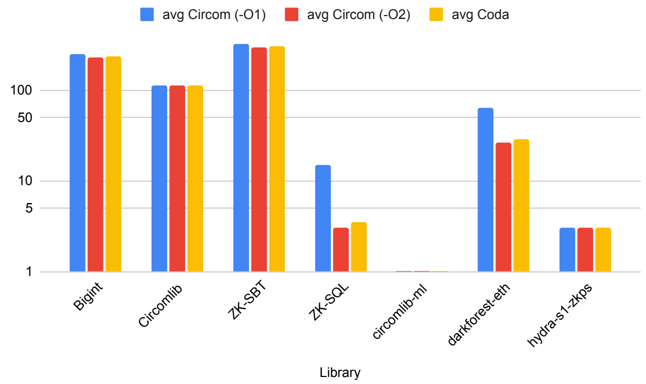

Figure 11. The quality of R1CS constraints generated by CODA compared to Circom.

the body of a circuit is encoded as a conjunction of Coq propositions.

We use the benchmarks from the Circom-bigint library as a case study. Table 10 compares the verification effort required by each approach. The "Proof Length" columns show the number of lines of Coq proof that is required to certify the program using CODA and shallow embedding, respectively. On average, CODA enables 60% reduction in proof size.

Result for RQ3: CODA reduced the verification effort by requiring 60% shorter proofs compared to the shallow embedding approach.

6.4. Quality of Compiled R1CS

Finally, we investigate whether the CODA compiler is able to generate R1CS constraints whose size is comparable to the Circom compiler. Because the size of a constraint system crucially determines the cost producing the zero-knowledge proofs from the constraint system[1], a DSL that does not produce competitive R1CS constraints is unlikely to be adopted. To this end, we compare the number of constraints produced by the CODA compiler vs the Circom compiler. We exclude benchmarks that have transitive dependencies outside the benchmark set, as the compilation procedure requires all referenced circuits be defined. Additionally, if a circuit contains a natural number parameter, we instantiate the circuit using the representative values for said parameter and take the average across all parameter values. We compare CODA's generated constraints against Circom's constraints produced by optimization levels **-O1** and **-O2**.

The results of this evaluation are summarized in Figure 11. The key takeaway is that the CODA compiler is able to produce R1CS constraints that are of comparable quality to the Circom compiler. In particular, compared to the Circom compiler on level **-O1**, CODA generates equal or fewer constraints for 100% of the benchmarks. Compared to the Circom compiler on level **-O2**, for 82% benchmarks, the number of CODA's generated constraints is within 10% of the number of Circom's constraints, while for 26% benchmarks, CODA produced equal or fewer constraints than the Circom compiler.

Result for RQ4: The CODA compiler can generate constraints whose quality is comparable to Circom.

7. Related Work

In this section, we survey prior work that is most closely related to our proposed approach.

Refinement types. There has been significant interest in using refinement types for formal verification [32], [33], [40], [41], [36]. Some example applications include proving the absence of integer overflows [36] and verification of memory safety [33]. Most of the prior techniques in this space keep type checking decidable and use an off-the-shelf SMT solver to discharge the constraints generated by the type checker. However, zero knowledge circuit verification is much more challenging than prior applications of refinement types, and existing SMT solvers do not provide adequate support for reasoning about polynomial equations over finite fields. Thus, in contrast to prior work, CODA outputs Coq propositions during type checking and uses a combination of automated tactics and manual intervention (where necessary) to enable formal verification of ZK circuits.

ZK programming languages and compilers. Due to the increasing demand for zero-knowledge proofs, there have been several new domain-specific languages (DSLs) targeting this domain. Example DSLs for constructing zk-SNARKS include Leo [12], Noir [5], Zokrates [15], Zinc [26], Snarky [28], Lurk [3], and CirC [29], for effectively constructing zkSNARKs [10]. In contrast to these prior efforts, the main focus of CODA is on *verifiability* of the circuit with respect to the specification.

While most languages, include CODA, are intended for zkSNARKs, there are also other programming languages targeting zkSTARKs (Scalable Transparent ARgument of Knowledge) [8]. For instance, Cairo [19], is a Turing complete language that allows general computation, and has recently become quite popular for zkSTARKs.

Formal methods & ZK. Writing correct zero-knowledge programs requires specialized domain expertise. Due to the importance of ZK circuits in blockchain applications, recent work has proposed techniques for finding bugs in this domain [39], [4], [16], [38]. For instance, Circomspect [14] is an open-source static analyzer designed to find bugs in Circom programs. Circomspect performs syntactic pattern matching on the Circom source code to detect potential issues. However, this approach suffers from both false positives as well as false negatives and is intended as a bug finder rather than a verifier. Wen et al. [43] study common vulnerabilities in Circom and describe a static analysis framework for detecting these vulnerabilities. Their technique operates over an abstraction called the *circuit dependence graph (CDG)* that captures key properties of the circuit and allows expressing *semantic vulnerability patterns* as queries over the CDG abstraction. Another recent effort [30] focuses on finding underconstrained bugs (or proving the absence thereof) in Circom programs. This approach combines SMT solving with lightweight uniqueness inference to effectively reason about underconstrained circuits. CODA focuses on verifying the functional correctness of ZK programs. Unlike previous approaches that focus on finding common vulnerabilities, the focus of CODA is on verifying full correctness of the circuit with respect to a formal specification.

Another orthogonal line of work [25] combines formal methods and the zero-knowledge domain in different way. In particular, recent work has proposed ZK-UNSAT, a technique for proving that a propositional formula is unsatisfiable while revealing minimal information about the resolution refutation proof of unsatisfiability. As the focus of this work is generating zero-knowledge arguments of unsatisfiability, it is not directly related to CODA.

Formal verification for cryptography. There is a body of work on applying formal verification techniques to cryptographic protocols. For instance, FiatCrypto [17] introduces a new approach for implementing cryptographic arithmetic in high-level code with machine-checked proofs of functional correctness through Coq. Corin et al. [13] leverage a variant of probabilistic Hoare logic to prove the security of ElGamal; Gagne et al. [18] use similar methods to prove the security of the front-end of many CBC-based MACs, PMAC and HMAC. Tiwari et.al [37] leverage component-based program synthesis to automatically generate padding-based encryption schemes, and block cipher modes of operations. EasyCrypt [6] is a toolset that allows users to specify and prove the correctness of cryptographic protocols.

On the other hand, there is little work on reasoning about the correctness of zero-knowledge proofs. Leveraging the MPC-in-the-head paradigm (i.e., Multi-Parity Computation), Sidorenco et al.[35] generated machine-checked proofs of ZK protocols using EasyCrypt. Almeida et al. [2] developed a certifying compiler for $\Sigma-$protocols, a broad class of zero-knowledge protocols including zkSNARKs [10]. Specifically, given a protocol written in its high-level language, the compiler generates an executable implementation that is provably correct using the Isabelle/HOL [27] theorem prover. More recent work has focused on building *specialized solvers* for polynomial equations over finite fields [21], [30]. Similar to the custom Coq tactic library that reduces the proof burden in CODA, we believe the above-mentioned works are complementary and can be incorporated into CODA to improve the degree of proof automation.

8. Conclusion

We presented CODA, a new statically-typed language for building zero-knowledge applications. CODA enables developers to *formally specify* and *statically check* properties of ZK circuits through a rich refinement type system. The type checker underlying CODA generates a set of Coq lemmas that, if valid, collectively constitute a proof of correctness of the circuit. CODA facilitates interactive verification of these lemmas using a combination of domain-specific proof tactics and manual intervention where necessary. Finally, the CODA compiler automatically translates high-level CODA circuits to low-level R1CS constraints from which a zero-knowledge proof system can be generated using existing techniques.

We used CODA to formally specify and verify 79 existing circuits from widely-used Circom projects. We were able to verify the correct circuits and identified 6 previously unknown vulnerabilities in the original Circom circuits.

Appendices

8.1. Operational Semantics

The full operational semantics for CODA is shown in Figure 12.

8.2. Metatheory

In this section, we include some of the metatheory about CODA's type system as well as a brief sketch of the proof of theorem 1.

For the purposes of the proofs, we consider a core subset of the calculus that contains only field elements, integers, booleans, pairs (products of arity 2), functions, and iter expressions. Furthermore, we take "unit" to be a first-class expression unit and type Unit.

8.2.1. Preliminary Definitions.

Definition 1. The valuation typing relation, written as $\Delta; \Gamma \vDash \sigma$, is defined as:

$$\begin{aligned}
&\Delta; \Gamma \vDash \sigma ::= \\
&\quad \mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma) \\
\wedge &\quad (\forall x, v, \tau. \\
&\qquad \Gamma(x) = \{\nu : T \mid \phi\} \wedge \sigma(x) = v \implies \\
&\qquad\quad \Delta; \Gamma \vdash v : \{\nu : T \mid \nu = x\}) \\
\wedge &\quad (\forall x, v, x', \tau_1, \tau_2. \\
&\qquad \Gamma(x) = x' : \tau_1 \to \tau_2 \wedge \sigma(x) = v \implies \\
&\qquad\quad \Delta; \Gamma \vdash v : x' : \tau_1 \to \tau_2)
\end{aligned}$$

The meaning of valuation typing is "a variable x is in Γ if and only if it is also in σ; and the value v of every variable x in σ must have the same type as x under Γ." Note that the two "cases" here correspond to the types in the TE-VAR and TE-VAR-FUN rules, respectively.

Definition 2. The base type subtyping judgment, written as $\vDash T_1 <: T_2$, is defined using the inference rules in fig. 13.

8.2.2. Subtyping Lemmas. We now state several useful properties about our subtyping rules.

Lemma 1 (Every Refinement Type is a Subtype of True)**.** *For all* Γ*,* T*, and* ϕ*,*

$$\Gamma \vDash \{\nu : T \mid \phi\} <: \{\nu : T \mid \mathsf{true}\} \tag{1}$$

Proof. By induction on the derivation of eq. (1). □

Lemma 2 (Inversion for Function Subtyping, Unknown Supertype)**.** *If* $\Gamma \vDash x : \tau_1 \to \tau_2 <: \tau$*, then there exist* τ_1', τ_2' *such that* $\tau = x : \tau_1' \to \tau_2'$ *and* $\Gamma \vDash \tau_1' <: \tau_1$ *and* $\Gamma \vDash \tau_2 <: \tau_2'$.

Proof. By induction on the derivation of $\Gamma \vDash x : \tau_1 \to \tau_2 <: \tau$. □

Lemma 3 (Inversion for Function Subtyping, Both Known)**.** *If* $\Gamma \vDash x : \tau_1 \to \tau_2 <: y : \tau_1' \to \tau_2'$*, then* $x = y$ *and* $\Gamma \vDash \tau_1' <: \tau_1$ *and* $\Gamma \vDash \tau_2 <: \tau_2'$.

Proof. By induction on the derivation of $\Gamma \vDash x : \tau_1 \to \tau_2 <: y : \tau_1' \to \tau_2'$. □

Lemma 4 (Inversion for Refinement Subtyping)**.** *If* $\Gamma \vDash \{\nu : T_1 \mid p\} <: \tau$*, then there exists a* T_2'*,* q *such that all of the following hold:*

- $\tau = \{\nu : T_2 \mid q\}$
- $\Gamma \vDash T_1 <: T_2$
- $\vDash_{\mathsf{Coq}} \forall \vec{x} \in \mathsf{dom}(\Gamma).\ \forall \nu.\ \mathsf{Encode}(\Gamma) \implies [\![p]\!] \implies [\![q]\!]$

Proof. By induction on the derivation of $\Gamma \vDash \{\nu : T_1 \mid p\} <: \tau$. □

Lemma 5 (Subtyping Weakening)**.** *If* $\Delta; \Gamma \vDash \tau_1 <: \tau_2$ *and* $x \notin \mathsf{dom}(\Gamma)$*, then* $\Delta; \Gamma, x : \tau_x \vDash \tau_1 <: \tau_2$.

Proof. By induction on the derivation of $\Delta; \Gamma \vDash \tau_1 <: \tau_2$. □

8.2.3. Expression Typing Lemmas.

Lemma 6 (Expression Typing Weakening)**.** *If* $\Delta; \Gamma \vdash e : \tau$ *and* $x \notin \mathsf{dom}(G)$*, then* $\Delta; \Gamma, x : \tau_x \vdash e : \tau$

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e : \tau$. □

Lemma 7 (Inversion for Types)**.** *Suppose that* $\Gamma \vdash e : \tau$*. Then there exists a type* τ' *such that:*

$$\Delta; \Gamma \vdash e : \tau' \tag{2}$$

$$\Gamma \vDash \tau' <: \tau \tag{3}$$

and:

(a) *If* $e = f$ *for some* $f \in \mathbb{F}_p$*, then* $\tau' = \{\nu : \mathsf{F} \mid \nu = f\}$.

(b) *If* $e = b$ *for some* $b \in \{\mathsf{true}, \mathsf{false}\}$*, then* $\tau' = \{\nu : \mathsf{Bool} \mid \nu = b\}$.

(c) *If* $e = x$*, then exactly one of the following is true:*
- *There exist* T, ϕ *such that* $\tau' = \tau' = \{\nu : T \mid \nu = x\}$ *and* $\Gamma(x) = \{\nu : T \mid \phi\}$.
- *There exist* $x', \tau_1, \tau_2, \tau_1', \tau_2'$ *such that* $\tau = x' : \tau_1 \to \tau_2$ *and* $\Gamma(x) = x' : \tau_1' \to \tau_2'$ *and* $\Delta; \Gamma \vdash x : x' : \tau_1' \to \tau_2'$.

(d) *If* $e = \star$*, then* $\tau' = \{\nu : \mathsf{F} \mid \mathsf{true}\}$.

(e) *If* $e = \mathsf{Closure}(\lambda x : \tau_1'.\ e',\ \sigma)$ *for some* e', τ_1'*, then there exist* $\tau_1, \tau_2, \tau_2', \Gamma'$ *such that* $\tau = x : \tau_1 \to \tau_2$ *and* $\tau' = x : \tau_1' \to \tau_2'$ *and* $\Delta; \Gamma', x : \tau_1' \vdash e : \tau_2'$ *and* $\mathsf{FV}(\tau_2') \subseteq \{\nu, x\}$.

(f) *If* $e = \lambda x : \tau_1.\ e'$*, then there exists* e, τ_2 *such that* $\Delta; \Gamma \vdash (\lambda x : \tau_1.\ e) : x : \tau_1 \to \tau_2$.

(g) *If* $e = e_1\ e_2$*, then there exist* x, τ_1', τ_2' *such that* $\tau' = \tau_2'[e_2 \mapsto x]$ *and* $\Delta; \Gamma \vdash e_1 : x : \tau_1' \to \tau_2'$ *and* $\Delta; \Gamma \vdash e_1\ e_2 : \tau'$.

$$
\begin{array}{lcll}
v & ::= & & \textbf{Values:} \\
 & | & c & \text{constants} \\
 & | & (v_1, \ldots, v_n) & \text{product value} \\
 & | & v_1 :: v_2 & \text{array value} \\
 & | & \mathsf{Closure}(\lambda x : \tau.\ e,\ \sigma) & \text{closure} \\
\sigma & ::= & & \textbf{Valuation:} \\
 & | & \varnothing & \text{empty valuation} \\
 & | & \sigma[x \mapsto v] & \text{augmentation}
\end{array}
$$

$$\frac{}{\sigma \vdash v \Downarrow v}\ \textsc{E-Value} \qquad \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v}\ \textsc{E-Var}$$

$$\frac{v \in \mathbb{F}}{\sigma \vdash \star \Downarrow v}\ \textsc{E-NonDet} \qquad \frac{\sigma \vdash e_1 \Downarrow f \qquad \sigma \vdash e_2 \Downarrow f \qquad f \in \mathbb{F}}{\sigma \vdash \mathsf{assert}\ e_1 = e_2 \Downarrow \mathsf{unit}}\ \textsc{E-Assert}$$

$$\frac{\sigma \vdash e_1 \Downarrow f_1 \qquad \sigma \vdash e_2 \Downarrow f_2 \qquad f_1 \otimes f_2 = f}{\sigma \vdash e_1 \otimes e_2 \Downarrow f}\ \textsc{E-FBinop} \qquad \frac{\sigma \vdash e_1 \Downarrow z_1 \qquad \sigma \vdash e_2 \Downarrow z_2 \qquad z_1 \oplus z_2 = z}{\sigma \vdash e_1 \oplus e_2 \Downarrow z}\ \textsc{E-IntBinop}$$

$$\frac{\sigma \vdash e_1 \Downarrow b_1 \qquad \sigma \vdash e_2 \Downarrow b_2 \qquad b_1 \odot b_2 = b}{\sigma \vdash b_1 \odot b_2 \Downarrow b}\ \textsc{E-BoolBinop}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2 \qquad v_1, v_2 \in \mathbb{F} \text{ or } v_1, v_2 \in \mathbb{Z} \text{ or } v_1, v_2 \in \mathbb{B} \qquad v_1 = v_2}{\sigma \vdash e_1 = e_2 \Downarrow \mathsf{true}}\ \textsc{E-EqTrue}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2 \qquad (v_1, v_2) \notin \oslash}{\sigma \vdash e_1 \oslash e_2 \Downarrow \mathsf{false}}\ \textsc{E-EqFalse}$$

$$\frac{\sigma \vdash e_1 \Downarrow z_1 \qquad \sigma \vdash e_2 \Downarrow z_2 \qquad v_1, v_2 \in \mathbb{Z} \qquad \oslash \in \{<, \leq\} \qquad (z_1, z_2) \in \oslash}{\sigma \vdash e_1 \oslash e_2 \Downarrow \mathsf{true}}\ \textsc{E-IneqTrue}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2 \qquad v_1, v_2 \in \mathbb{Z} \qquad \oslash \in \{<, \leq\} \qquad (z_1, z_2) \notin \oslash}{\sigma \vdash e_1 \oslash e_2 \Downarrow \mathsf{false}}\ \textsc{E-IneqFalse}$$

$$\frac{}{\sigma \vdash \lambda x : \_.\ e \Downarrow \mathsf{Closure}(\lambda x : \_.\ e,\ \sigma)}\ \textsc{E-Lam} \qquad \frac{\sigma \vdash e_1 \Downarrow \mathsf{Closure}(\lambda x : \_.\ e,\ \sigma') \qquad \sigma \vdash e_2 \Downarrow v \qquad \sigma'[x \mapsto v] \vdash e \Downarrow v'}{\sigma \vdash e_1\ e_2 \Downarrow v'}\ \textsc{E-App}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \sigma \vdash e_2 \Downarrow v_2}{\sigma \vdash e_1 :: e_2 \Downarrow v_1 :: v_2}\ \textsc{E-Cons} \qquad \frac{\sigma \vdash e_1 \Downarrow v_h :: v_t \qquad \sigma \vdash e_2 \Downarrow 0}{\sigma \vdash e_1\, [e_2] \Downarrow v_h}\ \textsc{E-IndConsHead}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_h :: v_t \qquad \sigma \vdash e_2 \Downarrow z \qquad 0 < z \qquad \sigma \vdash v_t\, [z - 1] \Downarrow v}{\sigma \vdash e_1\, [e_2] \Downarrow v}\ \textsc{E-IndConsTail}$$

$$\frac{\sigma \vdash e_i \Downarrow v_i \text{ for all } i \in \{1, \ldots, n\}}{\sigma \vdash (e_1, \ldots, e_i) \Downarrow (v_1, \ldots, v_i)}\ \textsc{E-ProdCons} \qquad \frac{\sigma \vdash e_1 \Downarrow (v_1, \ldots, v_n) \qquad \sigma[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] \vdash e_2 \Downarrow v}{\sigma \vdash \mathsf{match}\, e_1\, \mathsf{with}\, (x_1, \ldots, x_n) \to e_2 \Downarrow v}\ \textsc{E-ProdDestr}$$

$$\frac{\begin{array}{c}\sigma \vdash e_l \Downarrow v_l \quad \text{for all } l \in \{s, e, f, a\} \\ v_s \in \mathbb{Z} \qquad v_e \in \mathbb{Z} \qquad v_s \geq v_e\end{array}}{\sigma \vdash \mathsf{iter}_{\lambda i.\tau}\ e_s\ e_e\ e_f\ e_a \Downarrow v_a}\ \textsc{E-Iter0} \qquad \frac{\begin{array}{c}\sigma \vdash e_l \Downarrow v_l \quad \text{for all } l \in \{s, e, f, a\} \\ v_s \in \mathbb{Z} \qquad v_e \in \mathbb{Z} \qquad v_s < v_e \qquad \sigma \vdash \mathsf{iter}_{\lambda i.\tau}\ (v_s +_{\mathbb{Z}} 1)\ v_e\ v_f\ (v_f\ v_a) \Downarrow v\end{array}}{\sigma \vdash \mathsf{iter}_{\lambda i.\tau}\ e_s\ e_e\ e_f\ e_a \Downarrow v}\ \textsc{E-IterS}$$

$$\frac{\Delta(C) = \textbf{circuit}\ R\ (x_1 : T_1) \cdots (x_n : T_n)\ \to T\ \{e\} \qquad \sigma \vdash e_i \Downarrow v_i \quad \text{for all } i \in \{1, \ldots, n\} \qquad \sigma[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] \vdash e \Downarrow v}{\sigma \vdash \#C\ e_1 \cdots e_n \Downarrow v}\ \textsc{E-CRef}$$

Figure 12. Semantics of CODA program

$$\frac{}{\vDash T <: T}\ \textsc{BSub-Refl}$$

$$\frac{\vDash T_{11} <: T_{21} \qquad \vDash T_{12} <: T_{22}}{\vDash (T_{11}, T_{12}) <: (T_{21}, T_{22})}\ \textsc{BSub-Pair}$$

Figure 13. Base Type Subtyping Rules

(h) *If* $e = (e_1, e_2)$*, then there exists* $T_1, T_2, \phi, T_1', T_2', \phi_1, \phi_2$ *such that:*

$$\tau = \{\nu : (T_1, T_2) \mid \phi\} \tag{4}$$

$$\tau' = \{\nu : (T_1', T_2') \mid \phi_1[\nu \mapsto \nu.1] \wedge \phi_2[\nu \mapsto \nu.2]\} \tag{5}$$

$$\Gamma \vDash \tau' <: \tau \tag{6}$$

$$\Delta; \Gamma \vdash (e_1, e_2) : \tau' \tag{7}$$

$$\Delta; \Gamma \vdash e_1 : \{\nu : T_1' \mid \phi_1\} \tag{8}$$

$$\Delta; \Gamma \vdash e_2 : \{\nu : T_2' \mid \phi_2\} \tag{9}$$

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. As there are many similar cases, we show only some of the cases here.

- Case TE-ConstF: Then we have
$$e = f \tag{10}$$
$$\tau = \{\nu : \mathsf{F} \mid \nu = f\} \tag{11}$$
Choosing $\tau' = \tau$ discharges eq. (2) and clause (a), and then applying TSub-Refl discharges eq. (3).
- Cases TE-NonDet, TE-ConstBool, TE-Var, TE-Abs: Similar.
- Case TE-Sub: Then
$$\Gamma \vDash \tau'' <: \tau \tag{12}$$
$$\Delta; \Gamma \vdash e : \tau'' \tag{13}$$
The inductive hypothesis is:
$$\Gamma \vdash e : \tau'' \implies \tag{14}$$
$$\exists \tau'.$$
$$\Delta; \Gamma \vdash e : \tau' \wedge \tag{15}$$
$$\Gamma \vDash \tau' <: \tau'' \wedge \tag{16}$$
$$\text{all clauses (a) - (h)} \tag{17}$$
Clearly eq. (14) is satisfied. Thus eq. (2) and eq. (3) follow immediately (the latter through transitivity of eq. (16) and eq. (12)).
We then proceed by case analysis on e to show the remaining clauses.
  - Case $e = f$: Only clause (a) applies, and eq. (17) implies that as required,
$$\tau' = \{\nu : \mathsf{F} \mid \nu = f\} \tag{18}$$
  - Case $e = \star$: Only clause (d) applies, and eq. (17) implies that as required,
$$\tau' = \{\nu : \mathsf{F} \mid \mathsf{true}\} \tag{19}$$
  - The remaining cases are similar.

□

Lemma 8. *If* $\Delta; \Gamma \vdash e : \tau$*, then for all* x, τ'*, we have* $\Delta; \Gamma, x : \tau' \vdash e : \tau$.

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e : \tau$. □

8.2.4. Correctness Properties.

Theorem 1 (Expression Type Preservation)**.** *If* $\Delta; \Gamma \vdash e : \tau$ *and* $\sigma \vdash e \Downarrow v$ *and* $\Delta; \Gamma \vDash \sigma$*, then* $\Delta; \Gamma \vdash v : \tau$.

Proof. By induction on the derivation of $\sigma \vdash e \Downarrow v$, where we want to prove the property:

$$\forall \Gamma, \tau. \quad \Gamma \vdash e : \tau \implies \Gamma \vdash v : \tau$$

As there are many cases, we show only some of the cases here.

- Case E-Value: Follows immediately from the premises.
- Case E-NonDet: Then
$$e = \star \tag{20}$$
$$v = f \in \mathbb{F}_p \tag{21}$$
From lemma 7,
$$\Gamma \vDash \{\nu : \mathsf{F} \mid \mathsf{true}\} <: \tau \tag{22}$$
$$\Delta; \Gamma \vdash e : \{\nu : \mathsf{F} \mid \mathsf{true}\} \tag{23}$$
Applying TE-ConstF and lemma 1 to v then yields the desired result.
- Case E-Var: Then
$$e = x \tag{24}$$
$$v = \sigma(x) \tag{25}$$
By lemma 7, we know that τ is either a refinement type or a function.

  If τ is a refinement type, i.e. $\Gamma(x) = \{\nu : T \mid \phi\}$ for some T, ϕ, then we have that $\Gamma \vDash \{\nu : T \mid \nu = x\} <: \tau$. From definition 1, it follows that $\Gamma \vdash v : \{\nu : T \mid \nu = x\}$. The conclusion then follows by applying the TE-Sub rule.
  Otherwise, the τ is a function type, i.e. $\Gamma(x) = x' : \tau_1 \to \tau_2$ for some x', τ_1, τ_2. The conclusion follows similarly in this case.
- Case E-Assert: Then
$$e = (\mathsf{assert}\ x_1 = x_2) \tag{26}$$
$$\sigma \vdash x_1 \Downarrow f \tag{27}$$
$$\sigma \vdash x_2 \Downarrow f \tag{28}$$
$$v = \mathsf{unit} \tag{29}$$
In order to show that $\Delta; \Gamma \vdash \mathsf{unit} : \{\nu : \mathsf{Unit} \mid x_1 = x_2\}$, we can use the TE-Sub, TE-Unit, and TSub-Refine rules. This requires us to show that $x_1 = x_2$, when embedded as a Coq proposition, is entailed by the embedding of Γ:
$$\vDash_{\mathsf{Coq}} \forall \vec{x} \in \mathsf{dom}(\Gamma).\ \forall \nu.$$
$$\mathsf{Encode}(\Gamma) \implies \tag{30}$$
$$[\![\nu = \mathsf{unit}]\!] \implies [\![x_1 = x_2]\!]$$
We now argue that the above proposition holds. First, by lemma 7, we have that $\Delta; \Gamma \vdash x_1 : \{\nu : \mathsf{F} \mid \nu = x_1\}$ (similarly for x_2). Using the inductive hypotheses for

x_1 and x_2, it can then be shown, resp., that

$$\Delta; \Gamma \vdash f : \{\nu : \mathsf{F} \mid \nu = x_1\} \tag{31}$$
$$\Delta; \Gamma \vdash f : \{\nu : \mathsf{F} \mid \nu = x_2\} \tag{32}$$

Applying lemma 7 to the above then yields

$$\Gamma \vDash \{\nu : \mathsf{F} \mid \nu = f\} <: \{\nu : \mathsf{F} \mid \nu = x_1\} \tag{33}$$
$$\Gamma \vDash \{\nu : \mathsf{F} \mid \nu = f\} <: \{\nu : \mathsf{F} \mid \nu = x_2\} \tag{34}$$

By lemma 4, we must have

$$\begin{aligned} &\vDash_{\mathsf{Coq}} \forall \vec{x} \in \mathsf{dom}(\Gamma).\ \forall \nu. \\ &\qquad \mathsf{Encode}(\Gamma) \implies \\ &\qquad\quad [\![\nu = f]\!] \implies [\![\nu = x_1]\!] \end{aligned} \tag{35}$$
$$\begin{aligned} &\vDash_{\mathsf{Coq}} \forall \vec{x} \in \mathsf{dom}(\Gamma).\ \forall \nu. \\ &\qquad \mathsf{Encode}(\Gamma) \implies \\ &\qquad\quad [\![\nu = f]\!] \implies [\![\nu = x_2]\!] \end{aligned} \tag{36}$$

If we instantiate ν in both propositions by setting $\nu = f$, then together they will imply the single proposition:

$$\begin{aligned} &\vDash_{\mathsf{Coq}} \forall \vec{x} \in \mathsf{dom}(\Gamma). \\ &\qquad \mathsf{Encode}(\Gamma) \implies \\ &\qquad\quad [\![f = f]\!] \implies [\![f = x_1 \wedge f = x_2]\!] \end{aligned} \tag{37}$$

Finally, the above proposition implies that:

$$\begin{aligned} &\vDash_{\mathsf{Coq}} \forall \vec{x} \in \mathsf{dom}(\Gamma). \\ &\qquad \mathsf{Encode}(\Gamma) \implies \\ &\qquad\quad [\![\mathsf{true}]\!] \implies [\![x_1 = x_2]\!] \end{aligned} \tag{38}$$

This implies eq. (30), which completes the proof of this case.

- Case E-PAIR: Then we have

$$e = (e_1, e_2) \tag{39}$$
$$v = (v_1, v_2) \tag{40}$$

First, from lemma 7, we have

$$\tau = \{\nu : (T_1, T_2) \mid \phi\} \tag{41}$$
$$\tau' = \{\nu : (T_1', T_2') \mid \phi_1[\nu \mapsto \nu.1] \wedge \phi_2[\nu \mapsto \nu.2]\} \tag{42}$$
$$\Gamma \vDash \tau' <: \tau \tag{43}$$
$$\Delta; \Gamma \vdash (e_1, e_2) : \tau' \tag{44}$$
$$\Delta; \Gamma \vdash e_1 : \{\nu : T_1' \mid \phi_1\} \tag{45}$$
$$\Delta; \Gamma \vdash e_2 : \{\nu : T_2' \mid \phi_2\} \tag{46}$$

We then apply the inductive hypotheses with e_1 and e_2 to obtain:

$$\Delta; \Gamma \vdash v_1 : \{\nu : T_1' \mid \phi_1\} \tag{47}$$
$$\Delta; \Gamma \vdash v_2 : \{\nu : T_2' \mid \phi_2\} \tag{48}$$

Finally, we apply TE-SUB and TE-PAIR to derive the conclusion.

- Cases E-FBINOP, E-BBINOP, E-INTBINOP: Similar.
- Case E-APP: Then

$$e = e_1\ e_2 \tag{49}$$
$$\sigma \vdash e_1 \Downarrow \mathsf{Closure}(\lambda x : \tau_1'.\ e',\ \sigma) \tag{50}$$
$$\sigma \vdash e_1 \Downarrow v_2 \tag{51}$$
$$\sigma[x \mapsto v_2] \vdash e' \Downarrow v \tag{52}$$

By lemma 7 on the type of e, there must exist $x', \tau_1, \tau_2, \tau_2'$ such that

$$\Delta; \Gamma \vdash e_1 : x' : \tau_1 \to \tau_2 \tag{53}$$
$$\Gamma \vDash x' : \tau_1' \to \tau_2' <: \tau \tag{54}$$
$$\Gamma \vDash \tau_2[x' \mapsto e_2] <: \tau \tag{55}$$

Applying the inductive hypothesis to e_1 then yields

$$\Delta; \Gamma \vdash \mathsf{Closure}(\lambda x : \tau'.\ e',\ \sigma) : x' : \tau_1' \to \tau_2' \tag{56}$$

Now, by applying lemma 7 to the closure, it follows that there exist Γ' and τ_2'' such that

$$\Delta; \Gamma \vdash \mathsf{Closure}(\lambda x : \tau.\ e',\ \sigma) : x' : \tau_1' \to \tau_2'' \tag{57}$$
$$\Gamma \vDash x' : \tau_1' \to \tau_2'' <: x' : \tau_1'' \to \tau_2'' \tag{58}$$
$$\Delta; \Gamma', x : \tau_1' \vdash e' : \tau'' \tag{59}$$

The inductive hypothesis can then be applied to e' to obtain:

$$\Delta; \Gamma', x : \tau_1' \vdash v : \tau'' \tag{60}$$

Note that τ'' may only contain x' and ν as free variables; consequently, $\tau''[x' \mapsto e_2]$ contains only free variables in $\mathsf{dom}(\Gamma)$. Furthermore, because v is already well-typed under Γ' and has no free variables besides ν, it can be typed under any other typing context. It follows that $\Delta; \Gamma \vdash v : \tau''[x' \mapsto e_2]$. The desired result follows immediately by applying TE-SUB and eq. (55).

- Case E-LAM: Follows from lemma 7, TE-SUB, and TE-CLO.
- Cases E-CREF, E-ITER0, E-ITERS: Similar to E-APP.

□

References

[1] Elvira Albert, Marta Bellés-Muñoz, Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Distilling constraints in zero-knowledge protocols. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 430–443. Springer, 2022.

[2] José Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. volume 6345, pages 151–167, 09 2010.

[3] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Daniel Rogozin, Cameron Wong, et al. Lurk: Lambda, the ultimate recursive knowledge. *Cryptology ePrint Archive*, 2023.

[4] aztec. Disclosure of recent vulnerabilities. https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities, 2022.

[5] Aztec. Introducing noir. https://noir-lang.org/, 2022.

[6] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *FOSAD*, 2013.

[7] Marta Bellés-Muñoz, Jordi Baylina, Vanesa Daza, and José L. Muñoz-Tapia. New privacy practices for blockchain software. *IEEE Software*, 39(3):43–49, 2022.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, page 46, 2018.

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 781–796. USENIX Association, 2014.

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014. USENIX Association.

[11] The Zero Knowledge Blog. R1cs. https://www.zeroknowledgeblog.com/index.php/the-pinocchio-protocol/r1cs, 2022.

[12] Collin Chin. Leo: A programming language for formally verified, zero-knowledge applications. https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-leo.pdf, 2021.

[13] Ricardo Corin and Jerry den Hartog. A probabilistic hoare-style logic for game-based cryptographic proofs (extended version), 2005. To appear in ICALP 2006 Track C corin@cs.utwente.nl 13264 received 23 Dec 2005, last revised 26 Apr 2006.

[14] Fredrick Dahlgren. It pays to be circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 09 2022.

[15] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, 2018.

[16] electriccoin. Zcash counterfeiting vulnerability successfully remediated. https://tinyurl.com/3sv793fj, 2019.

[17] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219. IEEE, 2019.

[18] Martin Gagné, Pascal Lafourcade, and Yassine Lakhnech. Automated security proofs for almost-universal hash for mac verification. Cryptology ePrint Archive, Paper 2013/407, 2013. https://eprint.iacr.org/2013/407.

[19] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo - a turing-complete stark-friendly cpu architecture. *IACR Cryptol. ePrint Arch.*, 2021:1063, 2021.

[20] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985.

[21] Thomas Hader. Non-linear smt-reasoning over finite fields, 2022.

[22] Iden3. Snarkjs. https://github.com/iden3/snarkjs, 2018.

[23] INRIA. The coq proof assistant. https://coq.inria.fr/, 2022.

[24] Matter labs. The future-proof zkevm. https://zksync.io/, 2022.

[25] Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2203–2217. ACM, 2022.

[26] Matter-Labs. Zinc. https://github.com/matter-labs/zinc, 2022.

[27] Tobias Nipkow, Markus Wenzel, and Lawrence Charles Paulson. Isabelle/hol: A proof assistant for higher-order logic. 2002.

[28] o1 Labs. Snarky: Write efficient, beautiful, safe zk-snark code. https://o1-labs.github.io/snarky/, 2022.

[29] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. Cryptology ePrint Archive, Paper 2020/1586, 2020. https://eprint.iacr.org/2020/1586.

[30] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Gaffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. Automated detection of underconstrained circuits for zero-knowledge proofs. *Cryptology ePrint Archive*, 2023.

[31] Polygon. Bring ethereum to everyone. https://polygon.technology/polygon-zkevm, 2022.

[32] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery.

[33] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 131–144, New York, NY, USA, 2010. Association for Computing Machinery.

[34] Scroll. The native zkevm scaling solution for ethereum. https://scroll.io/, 2022.

[35] Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters. Formal security analysis of mpc-in-the-head zero-knowledge protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–14, 2021.

[36] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. Soltype: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.

[37] Ashish Tiwari, Adria Gascon, and Bruno Dutertre. Program synthesis using dual interpretation. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *LNCS*, pages 482–497, 2015.

[38] TornadoCash. Tornado.cash got hacked. by us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8, 12 2019.

[39] trailofbits. The frozen heart vulnerability in bulletproofs. https://tinyurl.com/2z7nw5ku, 2022.

[40] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 269–282, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 310–325, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Veridise. Circom-pairing: A million-dollar zk bug caught early. https://tinyurl.com/3u5973pf, 2023.

[43] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. *Cryptology ePrint Archive*, 2023.