Evan Laufer emlaufer@cs.stanford.edu Stanford University

Alex Ozdemir aozdemir@cs.stanford.edu Stanford University

Dan Boneh dabo@cs.stanford.edu Stanford University

ABSTRACT

Interactive theorem provers (ITPs), such as Lean and Coq, can express formal proofs for a large category of theorems, from abstract math to software correctness. Consider Alice who has a Lean proof for some public statement T. Alice wants to convince the world that she has such a proof, without revealing the actual proof. Perhaps the proof shows that a secret program is correct or safe, but the proof itself might leak information about the program's source code. A natural way for Alice to proceed is to construct a succinct, zeroknowledge, non-interactive argument of knowledge (zkSNARK) to prove that she has a Lean proof for the statement T.

In this work we build zkPi, the first zkSNARK for proofs expressed in Lean, a state of the art interactive theorem prover. With zkPi, a prover can convince a verifier that a Lean theorem is true, while revealing little else. The core problem is building an efficient zkSNARK for dependent typing. We evaluate zkPi on theorems from two core Lean libraries: stdlib and mathlib. zkPi successfuly proves 57.9% of the theorems in stdlib, and 14.1% of the theorems in mathlib, within 4.5 minutes per theorem. A zkPi proof is sufficiently short that Fermat could have written one in the margin of his notebook to convince the world, in zero knowledge, that he proved his famous last theorem.

Interactive theorem provers (ITPs) can express virtually all systems of formal reasoning. Thus, an implemented zkSNARK for ITP theorems generalizes practical zero-knowledge's interface beyond the status quo: circuit satisfiability and program execution.

INTRODUCTION 1

Many popular applications, such as Zoom and Microsoft Office, are proprietary and their source code is not public. Such software is usually programmed in a high-level language but distributed as an opaque executable. Users download and execute it, with no hard guarantees of what it will do their computers. The user trusts that the software will work as intended.

For open source software, users can audit (or trust a third party to audit) the source directly. Using verification tools, one could even prove that the software is safe. Proprietary software precludes such audits or verification, but it allows the creator to protect intellectual property (IP) contained in the software's source. Essentially, proprietary software prioritizes IP protection over public auditability.

Conference'17, July 2017, Washington, DC, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-x-xxxx-x/YY/MM

https://doi.org/10.1145/nnnnnnnnnnnn

Is it possible to protect IP while also guaranteeing arbitrarily complex notions of safety? In theory, yes, by combining verification with cryptography. First, the developer would use an interactive theorem prover (ITP), such as Coq [1] or Lean [2], to define and verify the safety of their executable. Here the definition of safety is an ITP theorem and verification produces a proof of that theorem. However, publishing the ITP proof might leak information about the program's design. Such leakage occurs because an ITP proof about a program often relies on program invariants that are closely tied to its design [3]. Fortunately, a cryptographic proof like a zkSNARK [4] eliminates such leakage. With a zkSNARK for a language L one can prove that a statement is in L, without revealing anything else. Moreover, the proof is short and verification is fast.

With a zkSNARK for the language of correct ITP theorems, one could show that an executable is safe, without revealing anything else about the executable. Users would download the executable, the safety theorem, and the zkSNARK proof, and check the proof before running the executable.¹ In this paper, we make progress towards this vision by building the first zkSNARK for ITP theorems.

The Challenge. Building an efficient zkSNARK for ITP theorems is challenging. Recall that an ITP provides a language for specifying a theorem (e.g., that some executable is safe) and tools for constructing a proof of the theorem. Then, the ITP uses the Curry-Howard correspondence [6]: it compiles the theorem and proof respectively to a type τ and a term t in a dependently-typed lambda calculus; the proof is valid if and only if *t* has type τ (see Sec. 2).

Unfortunately, it seems difficult to build an efficient zkSNARK for even a basic dependent type checker.

- First, t and τ are large recursive datatypes, and there is almost no prior work on efficient zkSNARKs for recursive datatypes.
- · Second, dependent type-checking algorithms are recursive since t's type depends on the types of its sub-terms. Efficiently modeling recursion in zkSNARKs is challenging. Naively, the circuit model forces shallow recursions to have the same cost as deep recursions [7], so costs grow exponentially in recursion depth.

Our Approach. We overcome both challenges. First, we represent recursive datatypes as nodes with child pointers in a random-access memory, optimized using recent developments in zkSNARKs for randomized circuits [8]. This memory can be de-duplicated by the prover, mimicking a hash-consing table [9]. Further, by having the prover pre-create every term needed during type-checking, these memories become read-only, enabling another optimization [10].

Second, we optimize recursive type-checking via inference trees. A type-checker's only explicit output is a boolean: whether t has type τ . However, if the output is true, then there exists an implicit tree of primitive inferences that shows it. Since each inference is locally checkable, we have the prover materialize the tree and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

¹Such a tool would could also check *automatically constructed* proofs. Our techniques focus on dependent typing, which is also the basis of proofs from SMT solvers [5].

the verifier check it, node-by-node. Results of previous checks are cached, so every inference is checked just once. This cache is essentially a memoization table for the type-checker's recursion.

Third, the materialized inference tree approach is especially effective because checking the validity of a provided tree is concretely easier than algorithmically constructing a valid tree.

An Implemented zkSNARK for Lean Theorems. We apply our techniques to a restriction of the type system underlying the ITP *Lean* [2]. We built a system, called zkPi, that proves knowledge of a proof for a Lean theorem. The prover's input is a Lean theorem T and also a Lean proof P for that theorem. Here P is a secret "witness" for why the theorem is correct. The prover's output is a succinct zkSNARK proof π that it knows a valid Lean proof P for T. The verifier's input is the Lean theorem T along with the succinct zk-SNARK proof π . The verifier outputs "accept" or "reject." The proof system is succinct: the size of the proof π and the verifyer's time are constant (independent of the size of the original Lean proof P).

Lean proofs use more than basic dependent typing; they also use inductive constructions, the foundation for proofs by induction. Consequently, we must extend our zkSNARK system to handle inductive constructions. However, here, the privacy that we obtain has one limitation: the set of *inductive declarations* used in the proof is revealed. Formally, we model this leak by including this set in the instance provided as input to the verifier. This small leak does not affect our applications.

We obtain additional concrete and asymptotic optimizations by a suitable data structure design and type system modifications.

Evaluation. We evaluate zkPi by proving theorems from the two core Lean libraries: stdlib and mathlib. Section 1.1 gives a few examples of the theorems in these libraries. Within 4.5 minutes (per theorem) on an 8-core machine, zkPi can write zkSNARKs for 57.9% of stdlib theorems and 14.1% of mathlib theorems. Some theorems would require more resources, while others use Lean constructs that we do not support. More work is needed to write zkSNARKs about complex program safety properties, but our results represent an important step towards that goal. Not only do we produce the first-ever zkSNARK for any Lean theorem, but *our system also works for a significant fraction of the two biggest Lean libraries.* We discuss directions for improvement in Section 9.

A Broader Perspective. A practical zkSNARK for ITP theorems narrows the gap between the theory and practice of zero-knowledge. In theory, there is a ZK proof system for *any language* in PSPACE [11]. However, *efficient, implemented* zkSNARKs support languages similar to circuit satisfiability [8, 12–25]. So, an efficient zkSNARK for another language requires an efficient reduction to circuit satisfiability. Historically, this has only been done for languages similar to circuit satisfiability—such as program execution [12, 26–37].

Recent work by Luo et al. [38]. extends the interface of practical (interactive) ZK by building a system for resolution proofs of propositional *un*satisfiability. Our practical zkSNARK for ITP theorems extends the interface even further. In the language of an ITP, one can express resolution proofs, DRAT proofs (unsatisfiability proofs than can be exponentially smaller than resolution [39]), and much more. In fact, ITPs can express virtually *all* human-designed systems of formal reasoning. Of course, not all ITP theorems are provable in practice, but for those that are, we would like a practical zkSNARK. A sufficiently efficient zkSNARK for ITP theorems would create a world where "Everything [Practically] Provable is [Practically] Provable in Zero-Knowledge" [11].

Contributions. In sum, our contributions are:

- 1. Techniques for building a zkSNARK of dependent typing; e.g.:
 - a representation of lambda calculus terms (§4.1)
 - the use of materialized inference trees (§4.2)
- 2. Techniques for zkSNARKs of dependent typing with inductive constructions (§4.4).
- 3. Data-structure and type system optimizations (§4.3-§4.8).
- 4. The first implemented zkSNARK for Lean theorems; The system successfully proves knowledge of a Lean proof for a significant fraction of the theorems in stdlib and mathlib (§6–§7).

1.1 Examples

Mathematical Theorems. Figure 1 shows three mathematical theorems that we proved with zkPi, and one that we have not proved. Figure 1a is the commutativity of logical and: for all propositions *a*, *b* we have that $a \wedge b$ holds iff $b \wedge a$ does. Our proof is an explicit Lean term. The next theorem (Fig. 1b) is the handshaking lemma: in an undirected graph, the sum of vertex degrees is even. Our proof is inductive, and is written in Lean's tactic language. The third theorem (Fig. 1c) is a statement of the pigeonhole principle. Informally: if *n* pigeons are placed into n + 1 holes, then there must be one hole with two pigeons. We formalize this as an equivalent Lean statement: a list of natural numbers whose sum is greater than its length must have an element that is greater than one. Here, the proof is more complex than the previous examples (see App. D). Lastly, we state the Collatz Conjecture in Figure 1d. Although the statement is very simple, there is no known proof. However, using zkPi, one would be able to succinctly show they had found such a proof in zero-knowledge. In fact, zkPi proofs are sufficiently short that they would fit in Fermat's proverbial margin.

Properties about Programs. Figure 1e shows a function dedup_list which accepts a list of natural numbers and returns a list with no duplicates. The function is correct if, given list *ol*, the output list *l*: has no duplicates, *l* contains all values in *ol*, and *ol* contains all values in *l*. We prove this function correct. Since the type signature encodes correctness (but not the implementation), zkPi is able to prove dedup_list is correct without revealing the implementation. One can also state and prove properties of programs in languages other than Lean. This is done by encoding the semantics of the language in Lean, and representing the program as a lean term [41].

2 BACKGROUND

2.1 zkSNARKs

An argument of knowledge [4] for a predicate ϕ involves a prover \mathcal{P} and a verifier \mathcal{V} . \mathcal{V} knows an *instance* x and a predicate ϕ . \mathcal{V} asks \mathcal{P} to show that it knows a *witness* w such that $\phi(x, w)$ holds. To do so, \mathcal{P} constructs a short and efficiently checkable *proof* π . A *non-interactive argument of knowledge* comprises three algorithms:

- Setup(ϕ) \rightarrow (pk, vk): a pre-processing algorithm
- Prove(pk, x, w) $\rightarrow \pi$: creates a proof that $\phi(x, w) = \top$

```
theorem and.comm
                                                 theorem degree_is_even (V: Type u) (g:
                                                                                                  theorem pigeonhole
    (a b : Prop) : a \land b \leftrightarrow b \land a :=
                                                      @graph V) : even (degree g) :=
                                                                                                   (l : nat_list)
    iff.intro (and.swap) (and.swap)
                                                begin
                                                                                                   : ((list_len 1) < (sum_list 1))
                                                   induction g, trivial, exact g_ih,
                                                                                                      \rightarrow (double_pigeon 1)
           (a) The commutativity of \wedge
                                                   exact not_not_intro g_ih,
                                                                                                  • =
                                                                                                   /- ommitted -/
                                                 end
                                                            (b) Handshaking Lemma
                                                                                                                (c) Pigeonhole Principle
    inductive collatz : \mathbb{N} \rightarrow Prop
                                                                             structure deduped_list (ol : list ℕ) :=
1
                                                                          1
2
    | coll0 : collatz 0
                                                                          2
                                                                                (1 : list \mathbb{N}) (p : nodup 1)
    | coll1 : collatz 1
                                                                          3
                                                                                 (sub : subset 1 ol) (sub2: subset ol 1)
3
    | coll_even : \forall n : \mathbb{N}, collatz n \rightarrow collatz (2 * n)
4
                                                                          4
5
    | coll_odd : \forall n : \mathbb{N} , collatz (6 * n + 4) \rightarrow
                                                                          5
                                                                              def dedup_list \Pi (ol : list \mathbb{N}), deduped_list ol :=
6
                                collatz (2 * n + 1)
                                                                          6
                                                                                 /- omitted -/
7
8
    theorem collatz_conjecture : \forall n : \mathbb{N}, collatz n := sorry
                                                                              encoded within the type deduped_list.
```

(d) The Collatz Conjecture [40]

(e) A deduplication function. Notice the correctness conditions are

Figure 1: Examples of Lean theorems. Some details elided (see Appendix D). We have proven all (save collatz) with zkPi.

• Verify $(vk, x, \pi) \rightarrow \{\top, \bot\}$: accepts or rejects the proof

Two properties must hold: completeness and computational knowledge soundness. Informally, completeness means that if w is valid, then π must be valid and *computational knowledge soundness* means that it must be computationally infeasible to construct a valid π without knowing a valid w. See Appendix C for precise definitions.

Typically, proof systems do not operate directly on ϕ . Instead, a compiler converts ϕ into an intermediate representation (IR) for the proof system. Example IRs include Plonkish [42], QAPs [43], and AIR [44]. There are zkSNARKs for different IRs [8, 12-22, 26-37].

We build on systems that use arithmetic circuits (or similar) as the IR. A circuit C is a directed acyclic graph. It takes x and w as vectors of elements in a prime field \mathbb{F}_p , its gates compute + and ×, and it enforces some set of equalities. The circuit is satisfied if all equalities hold. The size of C, denoted |C|, is the number of non-linear multiplications (i.e., between two non-constants). A circuit compiler outputs: C, an instance encoder Enc_x and a witness encoder Enc_{w} such that $C(\operatorname{Enc}_{x}(x), \operatorname{Enc}_{w}(x, w)) \iff \phi(x, w)$.

We build on zero-knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs) [4]. Informally, zero-knowledge means that π must reveal nothing about w other than its existence. Suc*cinctness* means that $|\pi|$, |vk|, and verification time are polylogarithmic in |C| and linear in |x|. Note that proving time and memory can be quasi-linear in |C|. Proving costs are the bottleneck in most applications of zkSNARKs, so, minimizing |C| is useful.

One optimization technique is using non-deterministic "advice" from \mathcal{P} . Suppose that \mathcal{P} wants to show that $x \in \mathbb{F}_p$ is nonzero, using only field equalities. It suffices to show that $x^{p-1} = 1$ (by Fermat's little theorem, this holds iff $x \neq 0$). However, evaluating x^{p-1} requires $\Theta(\log p)$ multiplications. Instead, \mathcal{P} can provide $i \in \mathbb{F}$ such that ix = 1. If $x \neq 0$, setting $i = x^{-1}$ satisfies the constraint; otherwise, it is unsatisfiable. Using non-determinism (new data from \mathcal{P}) is a widespread optimization technique when using zkSNARKs.

Randomized circuits. Recent work [8] optimizes zkSNARKs further through *randomized* circuits. The circuit C is extended to also take as input a random $r \in \mathbb{F}^{n_r}$. Let $p_{C,x,w'}$ denote $\Pr[C(\operatorname{Enc}_x(x), w', r)]$: the probability that *C* accepts $(Enc_x(x), w')$. If $\phi(x, w)$ implies that $p_{C,x,Enc_w(x,w)} = 1$, then C is complete for ϕ . The soundness error ϵ_s is the least upper bound on $\{p_{C,x,w'}\}_{x,w'}$ when $\nexists w, \phi(x,w).$ If ϵ_s is negligible,² then C is also *sound*. Given a sound and complete C, one can build a zkSNARK for ϕ [8]. We discuss the precise security definition of a zkSNARK for a randomized circuit in Appendix C.

Randomization is a powerful optimization tool. While standard circuits capture the complexity class \mathcal{NP} , randomized circuits capture $\mathcal{M}\mathcal{A}$ (Merlin-Arthur) [45]. Importantly, some problems have smaller randomized circuits than deterministic ones. For instance, the smallest known deterministic circuit for checking whether two vectors $\vec{x}, \vec{y} \in \mathbb{F}^n$ are permutations of one another has size $\Theta(n \log n)$ [46]. However, consider a circuit that enforces $\prod_i (x_i - x_i)$ r) = $\prod_{i}(y_i - r)$, for random r. This circuit checks the same property with soundness error $n/|\mathbb{F}|$ —using only $\Theta(n)$ multiplications.

Hash Functions. We present two keyed hash functions. Let $k \in \mathbb{F}$ and $\vec{x} \in \mathbb{F}^n$. The root hash H_r [47] and coefficient hash H_c are defined:

$$H_{\rm r}(k,\vec{x}) \triangleq \prod_{i=0}^{n-1} (k-x_i) \qquad H_{\rm c}(k,\vec{x}) \triangleq \sum_{i=0}^{n-1} k^i x_i$$

Both are universal hash functions: they are collision resistant if the key k is chosen after the inputs \vec{x}, \vec{x}' . H_c distinguishes inputs that differ as sequences, while H_r distinguishes inputs as multisets. For both, the probability of a collision is $\leq n/|\mathbb{F}|$ and the probability of any pairwise collision among Q inputs is $\leq Q^2 n / |\mathbb{F}|$ [48].

Read-Only Memory. These hash functions can be used to build a zkSNARK for a circuit with random-access to a read-only memory (ROM). The ROM is a sequence of k values x_1, \ldots, x_k (private or public). The circuit's inputs are augmented with a list of Naccesses $(adr_i, val_i)_{i=1}^N$, each comprising an address and a value.

²A quantity *f* is negligible if for all $c \in \mathbb{N}$, $f = o(\lambda^{-c})$, with λ the security parameter.



Figure 2: An example tree with $C = (f \mapsto \Pi x : \mathbb{N}.\mathbb{N}, g \mapsto \Pi y : \mathbb{N}.\mathbb{N}, a \mapsto \mathbb{N}).$

$$\frac{C \vdash f : (\Pi x : A.B)}{C \vdash f : a : B} \xrightarrow{C \vdash a : A} (x \mapsto a, C) \vdash B \Downarrow B'$$

Figure 3: The judgement rule for typing an application.

The (extended) predicate holds if: the circuit is satisfied and for all $i \in \{1, ..., N\}$, $1 \le adr_i \le k$ and $x_{adr_i} = val_i$.

Previous work studies zkSNARKs for circuits with access to read-write and read-only memories [8, 26, 29, 47, 49, 50]. For ROMs with private data, the best existing solution [10] is based on H_r and H_c . It supports values x_i that are each field vectors of length- ℓ . Checking that the accesses are consistent with some ROM requires a (randomized) sub-circuit of size $O(\ell(N + k))$, with soundness error $\leq (1+\ell)(N+k)/|\mathbb{F}|$. Read-write memories are more expensive.

2.2 Theorem Proving and Dependent Typing

Lean [2] is an interactive theorem prover (ITP): a tool for specifying and proving theorems. Users write theorems and proofs in a high-level language. The language can also express programs, so Lean admits theorems (and proofs) about computer systems. The correctness of the proofs is checked by a small *kernel* module.

Lean's kernel reduces checking proofs to type-checking lambda calculus terms. The kernel comprises three algorithms:

- Thm2Type (maps a Lean theorem ϕ to a lambda-calculus type τ),
- Pf2Term (maps a Lean proof ρ to a lambda-calculus term t), and
- TypeCheck $(t, \tau) \rightarrow \{0, 1\}$.

The kernel is *sound* if: for all ϕ , if there exists a *t* such that TypeCheck(*t*, Thm2Type(ϕ)) = 1, then ϕ is true. The user has found a valid proof ρ for ϕ when TypeCheck(Pf2Term(ρ), Thm2Type(ϕ)) = 1.³

Lean's variant of the lambda calculus has a *dependent type system*. Recall that in the lambda calculus, function terms are written as λx . *e* (*x* is the argument and *e* is the body), and applications have form f x. In a dependent type system, functions have *Pi* type: written $\Pi x : X$. *Y*, where *X* is the type of the argument and *Y* is the return type. A Pi type may be *dependent*, meaning the resultant *type Y* can depend on the *value* of the argument *x*. As an example, If IntArray<N> is the type of length-N arrays, a function that takes an integer x and returns an array of x zeros has dependent type.

Since dependent typing allows types to depend on values, typechecking requires reasoning about *evaluation*. In fact, dependent type-checkers have three modules: typing, evaluation, and equality. The *typing* module reasons about whether a term *t* has type τ , written $t : \tau$. The *evaluation* module reasons about whether *t* reduces to *t'*, written $t \downarrow t'$ (e.g., through function application or *beta reduction*). The *equality* module reasons about whether *t* and *t'* are *definitionally equal*, written $t \equiv t'$; this is case if they are: syntactically equal, equal modulo variable renaming, etc.

Abstracting, we say that typing, evaluation, and equality are different *relations* that the type-checker comprehends. Generally, dependent type systems do not syntactically distinguish "types" from "terms"; thus all three relations are binary relations over terms. Furthermore, each relation depends on a *context* C that maps variable names to terms. The notation $C \vdash t : \tau$ denotes that *t* has type τ *in the context of* C. As we will see, contexts are used to reason about variable bindings in typing and evaluation.

A dependent type system is defined by *judgement rules* (synonymously, *inference rules*) for determining whether a relation holds. Each judgement rule assumes hypotheses and yields a conclusion. For example, Figure 3 shows the judgement rule for function applications. It says that if f has type $\Pi x : A.B$, a has type A, and Bevaluates to B' when $x \mapsto a$ is added to the context, then f a has type B' (note that f a denotes the application of f to a).

While a judgement rule is essentially a template, a *judgement* shows a statement about concrete terms, e.g., 4 : nat. Judgement rules can be recursive; the application typing rule is an example: it depends on two other typing relationships and one evaluation relationship. Because of this recursion, a relationship between two concrete terms is shown by a *tree* (really, a DAG) of judgements.

Figure 2 shows an example judgement tree. It proves that f(g a) has type \mathbb{N} in a context where f and g are functions from \mathbb{N} to \mathbb{N} . The internal nodes are judgements for the type of a function application. The leaves comprise variable typing judgements (which depend on a context lookup, i.e., $C[t'] \mapsto \tau'$) and constant evaluation judgements (e.g., $\mathbb{N} \Downarrow \mathbb{N}$). While an inference algorithm's only explicit output is a type τ such that $C \vdash t : \tau$ ($\tau = \mathbb{N}$ in the example), the algorithm implicitly constructs the whole judgement tree.

Summarizing, in a dependent type system, a typing relationship is proved by a tree of judgements about typing, evaluation, and equality. A type-checker is an algorithm that takes as input terms tand τ , outputs whether $t : \tau$. If the typing relationship holds, the type-checker constructs (sometimes implicitly) a tree showing that.

Calculus of Inductive Constructions. Lean's type system extends the *Calculus of Inductive Constructions (CIC)* [51]: a dependent type system that also includes inductive definitions. These allow for userdefined inductive (i.e., recursive) data types. For example, Figure 4 shows the Lean definition of the natural numbers (lines 1–3). There are two constructors: zero takes no arguments and returns a nat, while succ recursive takes a nat, returning the next nat.

Each inductive type automatically defines a *recursor* (lines 5–9). The recursor (like the Y combinator) facilitates defining recursive functions over inductive datatypes. Its first argument (motive) is the type of the recursive function. Subsequent arguments (*minor*)

³This is an instance of the *Curry-Howard correspondence* [6], which relates valid proof to well-typed terms.

```
inductive \mathbb{N} : Type
1
2
      | zero : ℕ
3
      | succ : \mathbb{N} \to \mathbb{N}
4
      protected eliminator ℕ.rec :
5
       \Pi(\text{motive: } \mathbb{N} \rightarrow \text{Type*}),
6
7
         (motive zero) \rightarrow
8
         (\Pi(n: \mathbb{N}), \text{ motive } n \rightarrow \text{motive } (\text{succ } n)) \rightarrow
9
         (\Pi(n: \mathbb{N}), motive n)
10
11
      def is_zero : \mathbb{N} \rightarrow \text{bool} := \mathbb{Q}\mathbb{N}.\text{rec}
                                        -- type: \mathbb N -> bool
12
       (\lambda(n: \mathbb{N}), bool)
13
       bool.tt
                                        -- case: zero
14
       (\lambda(n: \mathbb{N}) (p: bool), bool.ff) -- case: succ
15
16
      #eval is_zero zero
                                              -- bool tt
      #eval is_zero (succ zero) -- bool.ff
17
```

Figure 4: Lean definition of the nat inductive type, its recursor, and a definition of is_zero

premises) are functions that compute recursive cases. The final argument (*major premise*) is the argument to the recursive function.

For example, is_zero (lines 11–14) defines a function from naturals to booleans that detects zero. The motive is a Pi from nat to bool: signifying our function will return bool. The first minor premise (this one applies to the zero constructor) is just bool.tt. The second minor premise is recursive. It ignores its arguments and returns bool.ff.

The recursor can also be used to write inductive proofs over the datatype (e.g., induction over the natural numbers). This is possible, essentially, because motive can be dependent. In such a proof, there is one inductive case for each recursive constructor. Induction declarations can also be polymorphic over any number of type parameters. This allows for general-purpose container types like lists and maps. Typing and evaluation rules are added to the type system for all constructors and the eliminator.

Lean's type system includes other advanced features: type universes {Sort i}_{$i \in \mathbb{N}$}, definitional equivalence for eta expansions, proof irrelevance, quotient types, and more.

3 OVERVIEW: A ZKSNARK FOR LEAN THEOREMS

Figure 5 shows the interface and pseudo-code for our zkSNARK for Lean theorems. Setup accepts the security parameter and circuit size parameters as arguments, and outputs the prover and verifier keys. First, the function generates a circuit with the given parameters cp. These parameters include the (maximum) number of judgements, terms, axioms, inductive declarations, etc. Then, it uses CirC [7] and Mirage [8] to create the proving key pk and verifying key vk.⁴

Prove generates a proof π for a Lean theorem ϕ . It accepts pk and four Lean constructs: the proof ρ , theorem ϕ , axioms list ax, and a list of inductive declarations ind. ϕ , ax, and ind are public data. First,

Conference'17, July 2017, Washington, DC, USA

```
Setup(1^{\lambda}, cp) \rightarrow (pk, vk)

C \leftarrow CircuitGen(cp)

R \leftarrow CirC.Compile(C)

return Mirage.Setup(1^{\lambda}, R)

Prove(pk, \rho, \phi, ax, ind) \rightarrow \pi

t \leftarrow Simplify(Pf2Term(\rho))

\tau \leftarrow Simplify(Thm2Type(\phi))

x, w \leftarrow EncodeTermAndType(t, \tau, ax, ind)

return Mirage.Prove(pk, x, w)

Verify(vk, \pi, \phi, ax, ind) \rightarrow \{0, 1\}

\tau \leftarrow Simplify(Thm2Type(\phi))

x \leftarrow EncodeType(\tau, ax, ind)

return Mirage.Verify(vk, x, \pi)
```

Figure 5: Pseudo-code for zkPi: our zkSNARK for Lean

it uses Lean's exporting utilities (Pf2Term and Thm2Type) to encode ρ and τ as terms in the Calculus of Inductive Constructions (CIC). Then, it simplifies both of these using zkPi's simplification tool in an attempt to reduce their size (see Sec. 6). It encodes the simplified term and type using zkPi's encoder, which converts the term/type, axioms, and inductive declarations into the representation that the circuit *C* expects, and creates a tree of judgements which prove that the term is of the correct type. This tree is also a (private) circuit input. Lastly, this encoded input is given to the Mirage prover, which creates the final zkSNARK proof π .

Verify checks π 's validity using the verifying key vk, a Lean theorem ϕ , a list of axioms ax, and a list of inductive declarations ind. First, it encodes ϕ as a CIC term the Lean exporter and simplifies it. Then, it encodes the type, axioms, and inductive declarations in the representation that the circuit expects. Unlike Prove, it does not create a tree of judgements; those depend on ρ and t, which are private to the prover. Lastly, it verifies π using the Mirage verifier.

4 A ZKSNARK FOR DEPENDENT TYPING

In this section we describe the core part of zkPi and its optimizations. We consider the following problem: Let τ be a public CIC type that both the prover and verifier know. The prover would like to show that it knows a (secret) CIC term t such that the typing relation $\Gamma_0 \vdash t : \tau$ holds, where Γ_0 is an empty context. Thus, we must design an $\mathcal{M}\mathcal{A}$ verification circuit for the $\Gamma \vdash t : \tau$ relation.

The task suggests three immediate challenges:

- (1) As CIC terms, t and τ are recursive datatypes.
- (2) The definition of the typing relation is itself recursive—the type of term t may depends on the type of a sub-term t'—and other necessary relations are also recursive.
- (3) The typing relation involves operations on a data structure: the context Γ.

We begin by considering term representation (§4.1), recursive relations (§4.2), and contexts (§4.3) for a basic dependently-typed lambda calculus. Then, we consider advanced features and techniques: inductive constructions (§4.4), axioms (§4.6), and De Bruijn levels (§4.7). Finally, we discuss low-level optimizations (§4.8).

⁴Replacing Mirage with a different proof system (secure as defined in Appendix C) would be straightforward.

Evan Laufer, Alex Ozdemir, and Dan Boneh

	$ IIx : \langle t \rangle \langle t \rangle$ $ \langle t \rangle \langle t \rangle$ $ x$	l: usize , r: usize , } enum Kind { Lambda, Pi, App, Var, }
(a) Ab	stract syntax	(b) Concrete term structure



Throughout, we consider a circuit built in two stages. First, we describe the circuit in a high-level language that exposes non-circuit concepts such as structures, arrays, fixed-width integers, functions, control flow, etc. Second, a compiler converts this program into an \mathcal{MR} circuit. In our implementation, we specify a circuit design in the Z# language (which extends ZoKrates [36] v0.6.2) and compile it to a circuit with the CirC compiler infrastructure [7]. We implement some techniques in Z# and others as improvements to CirC.

4.1 Term representation

A term is a recursive datatype. Figure 6a shows the abstract syntax of terms: notice that there can be two subterms. We represent terms as nodes in memory: each term is a Z# structure with pointers to potential subterms (Fig. 6b). All terms reside in a global term array and that subterm pointers index into. Thus, accessing a subterm reduces to an array access; we discuss array accesses in Section 4.8.

We will use the same approach for other recursive datatypes in zkPi. Each is its own structure, in its own global array, with pointers into that array (and possibly the arrays of other datatypes). This mimics the memory layout created by a typed-arena allocator.

4.2 **Recursive relations**

Our goal is to enforce a typing relationship inside a zkSNARK. Recall that a type checker is given a term *t* and type τ . It determines whether $t : \tau$; if so it produces (often implicitly) a judgement tree. Thus, the type-checker *searches* for a valid judgement tree for $t : \tau$. This is more difficult than the corresponding decision problem: checking whether a given judgement tree is valid.

In a zkSNARK circuit we can optimize type-checking by materializing the judgement tree as prover advice. Then, the circuit only solves the decision problem: enforcing the validity of the tree. We represent judgements for that relation as a recursive datatype. The recursive arguments show that the judgements hypotheses hold. As with terms, we represent these recursive datatypes a Z# structures in a global array. Our circuit simply checks that all judgements are valid, and that the result is $t : \tau$, as desired.

Evaluation and definitional equality. The aforementioned judgement tree includes judgements that show evaluations and definitional equality, as well as typing. This is because the judgement rules for the different relations are mutually referential. For example, Figure 3 shows a typing rule that references evaluation, and Figure 7 shows a definitional equality rule that references typing. We handle all judgements in the same way: as recursive datatypes.

$$\frac{C \vdash a : \tau \qquad C \vdash b : \tau \qquad C \vdash \tau : Prop}{C \vdash a \equiv b}$$

Figure 7: Definitional equality holds for members of the same proposition. Prop refers to a distinguished Sort within the Lean type system, which we discuss in 4.5.

To optimize, we merge the relations for definitional equality and evaluation. That is, we make evaluation the union of the two relations. This reduces the number of cases in our judgement-checking circuit, shrinking the circuit. As we discuss in Section 5.1, this does not compromise soundness.

4.3 Contexts

Recall (Sec. 2.2) that contexts are crucial for evaluation and typing. In typing (respectively evaluation), for a term *t* containing a variable *x*, one must determine the type (resp., value) of *t* when *x* is bound to some type (resp., value) τ . We do this by including contexts in relations and judgements.

There is another approach for variable bindings, called *substitution*. In it, one eagerly replaces occurrences of x in t with τ , yielding a new term t'. A judgement about t under binding $x \mapsto \tau$ is instead applied to t'. Both approaches are correct, but perform differently.

While the Lean kernel uses substitution, we use contexts. Substitution creates many short-lived terms t'. Lean makes short-lived terms cheap using a custom allocator. However, it's unclear how to make short-lived terms cheap in a zkSNARK. As we will discuss (Sec. 4.8), memory in a zkSNARK is cheaper if it is read-only. But, read-only memory forces all objects to have the same (global) lifetime. So, it is not clear how to get substitution to perform well.

A context is a key-value map with insert and lookup. With nondeterminism, both operations reduce to a predicate $CtxElem(\Gamma', \Gamma, k, v)$, which holds when $\Gamma' = (\Gamma, k \mapsto v)$. To "look up" k in Γ', \mathcal{P} provides Γ . To "insert" $k \mapsto v$ into Γ, \mathcal{P} provides Γ' .

We reduce map operations to set operations using the coefficient hash (H_c , Sec. 2.1). The map becomes a set of hashes of key-value pairs. Naively, this allows one key to map to multiple values. This could let a malicious prover create invalid proofs using inconsistent substitutions. We prevent this by using De Bruijn levels to ensure that all binding names are fresh within an expression (§4.7).

Set representation. Typically, sets use comparisons or (keyless) hashing: both are expensive in our setting. Keyless hash functions are very non-linear, thus expensive in arithmetic circuits. Doing comparisons in arithmetic circuits (bitwise) is also expensive. We need a different kind of set.

We represent a set as a linked list, optimized with randomness and non-determinism. In a linked list, the first element is easily accessible. Consider a list *L* containing an item *x* that is *not* at the head of *L*. To access *x*, \mathcal{P} provides an equivalent list *L'* (i.e., one with the same elements as *L*) with *x* at the head. Then, \mathcal{P} convinces \mathcal{V} that *L'* and *L* are equivalent using the root hash function H_r (Sec. 2.1): $H_r(k,L) = H_r(k,L')$, for random *k*. Note if *L* and *L'* both have length less than N_C , the probability of a hash collision is at most $N_C/|\mathbb{F}|$, which is negligible.

This approach is efficient because evaluating H_r for L is cheap in an arithmetic circuit. Computing the hash of L from the hash of its tail requires one array access (to get the tail's hash) and one multiplication. Moreover, the hash computations can be amortized over all equivalence checks: one samples a single k, computes $H_r(k, \Gamma)$ for all contexts Γ , and uses those hash values to test equivalence.

We implement context list nodes (which contain a key, value, tail pointer, and hash) as Z# structures in a global array. One can show that H_r -collisions occur with negligible probability (Sec. 5.1).

Reducing Redundancy with Subcontexts. A naive implementation of context-based judgement-checking leads to *proof redundancy.* The root cause is that contexts can bind unused variables. Consider the (*) inferences in Figure 2:

$$(x \mapsto g \ a, C) \vdash \mathbb{N} \Downarrow \mathbb{N} \qquad (y \mapsto a, C) \vdash \mathbb{N} \Downarrow \mathbb{N}$$

For both inferences, the context is irrelevant. However, because the inferences syntactically contain distinct contexts, they are distinct and must be separately checked. Redundancy can occur even when the context *is* used. For instance, the judgement $C \vdash x \Downarrow \mathbb{N}$ only uses the value associated with variable *x* within *C*. It is syntactically distinct from $C' \vdash x \Downarrow \mathbb{N}$, even when C' also binds *x* to \mathbb{N} .

To reduce redundancy, we relax the type system. We allow the context for an antecedent judgement to be a *subset* of the context for the current judgement. The equivalence of this relaxation to the original type system follows from the strengthening property for CIC typing and evaluation We state the former here:

$$\frac{(\Gamma, x \mapsto A) \vdash e : \tau}{\Gamma \vdash e : \tau} \quad x \notin FV(e) \cup FV(\tau)$$

This optimization is only sound if context subsets bind *all* free variables in the expression being typed or evaluated. Otherwise, an evaluation $(\lambda x, x) y \downarrow x$ would incorrectly be allowed. In type checking, this is enforced in the variable typing rule: it requires the typed variable to have a context entry. In evaluation, we cannot require that all variables occur in the context (e.g. when evaluating $(\lambda x, x)$, the variable *x* never gets bound to a value). Instead, we enforce that the resultant expression correctly replaces substituted variables during lifting (§4.7), by requiring that the result of applications do not contain the variable just replaced.

Subcontexts can asymptotically reduce judgment tree size. Appendix A presents a family of CIC term-type pairs $\{(t_{\alpha}, \tau_{\alpha})\}$ where the tree size for $t_{\alpha} : \tau_{\alpha}$ is $\Theta(\alpha)$ with subcontexts but $\Theta(\alpha^2)$ without.

4.4 Inductive constructions

Recall (Sec. 2.2) that an inductive declaration creates a new datatype, constructors, and a recursor.

Representation. We represent inductive declarations themselves as public Z# structures that are passed into the circuit (Figure 8). We discuss the significance of this publicity in Section 5. Each constructor is referred to by an integer *name*. The declaration contains

- the number of type parameters,
- the number of recursive and non-recursive arguments for each constructor,
- a term pointer to the type of each constructor,
- a term pointer to the *recursor body*: the recursor without the type parameters or motive parameter, and

```
struct Ind {
   ty: usize,
   num_params: usize,
   ctors: [usize],
   num_nonrec_params: [usize],
   num_rec_params: [usize],
   rec_body: usize,
   rec_argc: usize,
   motive: usize,
}
```

Figure 8: Concrete inductive declaration structure

- the number of arguments the recursor expects,
- a term pointer to the *motive*,
- a term pointer to the inductive type itself.

Inductive types also require changes to term representation. Terms must now include inductive types, constructors, or recursors. First, we add three new entries to the enumeration of term kinds (Fig. 6b): IndType, IndCtor, and IndRec. Second, for terms, we add a (nullable) pointer to an inductive declaration structure (described in the next paragraph). For an inductive type, constructor, or recursor, this points to the inductive declaration. Finally, inductive constructor terms also contain the name of their constructor.

Evaluating Recursor Applications. Recursors introduce a new judgement rule for evaluating applications of the recursor to arguments. Figure 4 shows an example evaluation on line 17. Paraphrasing, the example reduction is

rec (λ n, bool) tt ret_ff (succ zero) \Downarrow ff

where ret_ff = $(\lambda n, p, ff)$. With this example, we will explain the five conditions required when evaluating a recursor application $t = \text{rec} \cdots$ (and how we ensure that they hold).

First, *t* must be well-typed, which we will discuss later.

Second, the *head* function of t must be the recursor. A non-applicative term is its own head; an applicative term's head is the head of the left pointer. In the example, the head is rec. We check that the head is correct for each term. To optimize, we cache each term's head in a new field of the term structure.

Third, the recursor must take the correct number of arguments. In the example, that is four. Like *head*, we check this for each term and cache it in the term structure.

Fourth, the final argument's head must be a constructor. In the example, that argument is succ zero (head: succ).

Fifth, the result must be an evaluation of an application of the correct minor premise. In the example, that minor premise is ret_ff. It must be applied to the arguments of the major premise (zero) and then the recursive evaluation. In sum, the result in our example is an evaluation of ret_ff zero (is_zero zero).

We identify and apply the correct minor premise with new relations get_arg and apply_elim.get_arg(f, i, g) holds when the i^{th} argument to f's head is g and apply_elim ensures that the application of the minor premise is well-formed (see App. B).

Other Considerations. Inductive type and constructor terms are simpler than recursor terms. Neither require an evaluation rule. Further, their typing rules are straightforward (App. B). However, inductive list {Type u} : Type u | list.nil : Π {T : Type u}, list T | list.cons : Π {T : Type u}, T \rightarrow list T \rightarrow list T

Figure 9: Lean definition for a generic list type, which uses a universe parameter *u* to support any inner type.

the typing rule for a recursor is somewhat complex because the recursor is universe polymorphic. We discuss further in Section 4.5.

Argument Constraints. To simplify the circuit, we only allow the final parameters of an inductive constructor to be recursive. Lean does not require this: for example, list.cons could accept the recursive list parameter first and then the value. However, our requirement is easy to satisfy: arguments could even be automatically re-ordered as a pre-precessing step.

We also do not support some types of recursive arguments. Lean allows recursive arguments to either appear by themselves (e.g. the second argument of list.cons) or as the body of a Pi-type (e.g. the argument can be a function which returns the inductive type). We do not support the latter; it would be more complex.

Inductive Families. An inductive type family is an inductive declaration that declares a family of types, instead of just a single type. Like for inductive types, their constructors can be recursive-i.e., can accept accepts a member of the inductive type family as a parameter. zkPi supports only non-recursive inductive type families (e.g. the eq inductive type family is supported, while vector is not). As a consequence, we also do not support mutually inductive types or nested inductive types, which are desugared into recursive inductive families. Adding recursive inductive type families would require modifications to the recursor evaluation judgements.

4.5 Universe Parameters

Here, *i* is called the *universe level*. Each type parameter specifies which sort it inhabits.

Monomorphizing Definitions. Lean allows definitions to be parameterized by (i.e., polymorphic over) universe levels. This allows for definitions that are polymorphic over the type hierarchy. Figure 9 shows an example of this used for the type list. Our circuit requires that all Lean theorems have their universe parameters resolved before being proven by the circuit. Essentially, the prover must monomorphize universe-polymorphic definitions. We require this to avoid implementing the complexity of universe polymorphism within the circuit.⁶ We believe this restriction does not bar many applications, and we discuss further in Section 9.

Polymorphic Recursors. However, we do allow for universe polymorphism in inductive recursors. We do this because it allows the

Evan Laufer, Alex Ozdemir, and Dan Boneh

prover to recurse to any type without specifying all the monomorphizations publicly (increasing privacy), and because it is simpler to implement than generic universe parameterization. For example, recall (Fig. 4) that the type of the recursor for nat is

nat.rec : $\Pi(m : nat \rightarrow Type*)$, (m zero) \rightarrow $(\Pi(n : nat), m n \rightarrow m (succ n)) \rightarrow$ $(\Pi(n : nat), m n)$

Here, Type* is syntactic sugar for any sort. Instead of allowing universe-polymorphic recursors, we could require the prover to publicly monomorphize them. However, allowing polymorphism is manageable for two reasons.

First, though a recursor may be polymorphic, its recursor body may not be. The recursor body refers to everything after the motive. For nat, this would be the term,

$$(m \text{ zero}) \rightarrow$$

 $(\Pi(n : nat), m n \rightarrow m (succ n)) \rightarrow$
 $(\Pi(n : nat), m n)$

Note that it is not universe polymorphic.

Second, the polymorphism imposes only a mild constraint when typing the recursor itself. Essentially, we allow the prover to claim the recursor has any suitable universe-monomorphized type. To elaborate on meaning of "suitable", we describe the four typing conditions for the recursor itself. First, the type must be a Pi type and have parameters for all the parameters of the inductive type. Second, the motive must be a Pi type which accepts the correct parameters, and returns some Sort. Third, the body of the recursor's type after the motive must be equal to the recursor body. This equality holds (even after monomorphizing) because the recursor body does not depend on the polymorphic universe level. Finally, certain recursors are not allowed to be universe polymorphic. Particularly, most inductive propositions are only allowed to have a motive which returns a Prop. We mark this restriction with an additional boolean In Lean, types are partitioned into different *universes* or *sorts*: {Sort i}_{$i \in \mathbb{N}$}.⁵ field in each inductive declaration, any_elim, which signifies the relation, well_formed_rec, described in Appendix B.

4.6 Axioms

An axiom is a type that is assumed to be inhabited; or, equivalently a theorem that is assumed to be true. Axioms are encoded in the circuit as a public input array of pointers to types. The axiom itself is also a kind of term. An Axiom term has kind Axiom, and contains a pointer into the axiom array. The typing judgement for an Axiom term follows the term pointer in the Axiom array, and ensures the type is correct. The axiom array should only contain pointers to public terms (or the prover could replace the term with whatever they like). But, we do not enforce this within the circuit, the verifier checks this constraint externally.

4.7 Lifting

Thus far we have described variable bindings using names. For example, $(\lambda x, b) e$ binds the variable named x to the expression e when evaluating b. However, we actually use De Bruijn levels to represent variable bindings. De Bruijn levels assign names to fresh bindings based on the number of previous bindings, where

 $^{^5 \}mathrm{Lean}$ distinguishes Sort 0 as the sort of $propositional \ types$ (e.g. $a \wedge b$). It is denoted Prop, and has special typing rules (see Appendix B).

⁶This complexity would be significant. Some typing rules (e.g., Type-Sort) modify universe levels, while others (e.g., Type-Pi) impose bounds on them (App. B). Thus, typing with universe polymorphism requires proving that such (symbolic) modifications respect (symbolic) bounds.

the most recent binding is referred to by adding one to the previous binding. These are similar to De Bruijn indices, where the binding names are reversed. For instance, in the term $(\lambda 0, \lambda 1, b)$, the first binding has name 0, and the second binding has name 1. This ensures that binding names are unique by construction, so alpha renaming and duplicate binding checks are not needed. We use De Bruijn levels instead of De Bruijn indices, because indices require re-indexing the context when adding a new binding. For example, when evaluating $(0 \mapsto e) \vdash (\lambda 0, 1) e')$, De Bruijn indices require incrementing the name of e within the context (0), to create a new context $(1 \mapsto e, 0 \mapsto e')$. In contrast, De Bruijn levels leave prior bindings intact. With levels, $(0 \mapsto e) \vdash (\lambda 1, 0) e'$ creates a new context $(0 \mapsto e, 1 \mapsto e')$.

De Bruijn levels necessitate *lifting*: re-indexing bound variables when substituting one term into another. For example, consider the term $(\lambda 0, \lambda 1, 0)(\lambda 0, 0)$. To evaluate it, we first add $(0 \mapsto (\lambda 0, 0))$ to the context. Then, to evaluate $(0 \mapsto (\lambda 0, 0)) \vdash (\lambda 1, 0)$, we substitute. Replacing 0 in $(\lambda 1, 0)$ with $(\lambda 0, 0)$ would invert the De Bruijn levels; instead, we lift the replacement to get $(\lambda 2, 2)$. Finally, we lift the substitution result $(\lambda 1, (\lambda 2, 2))$, to get $(\lambda 0, (\lambda 1, 1))$. We implement lifting as an additional relation within the circuit.

4.8 Additional optimizations

Memory. Our techniques require many accesses to large randomaccess memories that contain Z# structures. These memories are read-only because the objects they contain (terms, judgements, etc.) are checked—but not modified—by our circuit. Further, \mathcal{P} can ensure that all judgements are checked and all terms are used, so each address is accessed at least once. Thus, ROM optimizations apply (Sec. 2.1). We implemented them as CirC compiler passes; which allows our Z# circuit to be expressed in terms of array reads.

Relation Grouping. To hide the proportions of different judgement rules used in a proof, all judgements (evaluation, typing, ...) are structures in a single array, with a field that identifies the judgement rule. The array size is public, but this only reveals a bound on the total number of judgements.

This unification into a single array impacts performance. Suppose that rule *i* is checkable with a circuit of size s_i . The cost of checking that *some* rule has been correctly applied cannot be less than the *max*: max_i s_i (recall: circuits cannot branch). Every judgement pays this max_i s_i cost, even if that judgement's rule has cost $s_j \ll \max_i s_i$. This is particularly painful for lifting judgements, which are numerous, but very cheap to check on their own (Sec. 7.3).

To reduce the overhead of these rules, we split lifting judgements into their own array; their correctness is checked separately. If *l* is the number of lifting judgements, *p* is the number of nonlifting judgements, *s_l* is cost of checking lifting, and *s_p* is the cost of checking non-lifting, then this optimization reduces cost from $O((l + p) \max(s_l, s_p))$ to $O(ls_l + ps_p)$. However, it reveals an upper bound on the number of lifting rules used within the proof. As usual, this mild leakage can be mitigated by increasing the bound.

Node splitting. In our type system, some judgement rules (e.g., Eval-Ind, App. B) have four antecedent judgements, while most have two or fewer. Accessing each antecedent is an array access, so checking a judgement may require four accesses. Again, since

circuit cannot branch, these accesses are materialized for every judgement rule, *regardless of whether it actually uses the parent or not.* This means that the number of judgement rule lookups in a naive implementation with *p* judgement rules is 4*p*, even if all judgement rules only use two of the parents. To avoid this cost, we modify our type system. We split all judgement rules that have more than two parents into multiple rules, such that each individual rule only has two antecedent judgements. This modification is straightforward to implement. In the worst case (if all judgements require three lookups), this will introduce marginal overhead because of the redundant/unused fields within the intermediate judgement structures. However, in the best case (if all judgements require two lookups), this cuts the number of antecedent lookups by 50%.

5 SECURITY

5.1 Existential Soundness

First, we consider *existential soundness* of zkPi: whether it is feasible to construct a valid zkPi proof for a false Lean theorem ϕ . Here we sketch the security proofs, see Appendix E for further details. The existence of a proof ρ for ϕ reduces to whether the CIC type $\tau = \text{Thm}2\text{Type}(\phi)$ is inhabited by some term *t*:

$$\tau \in \mathcal{L}_{I} \qquad \mathcal{L}_{I} \triangleq \{\tau : \exists t, \Gamma_{0} \vdash t : \tau\}$$
(1)

Prior work on Lean's type theory shows that if (1) holds, then ϕ is true (relative to a reasonable background theory) [52, §6]. In our system, we establish (1) indirectly. We construct a randomized circuit $C(\tau, (t, w), r)$ that implements a modification of Lean's type system. That is, it respects a slightly different set \mathcal{L}'_{I} of inhabited types. Then, we show that C is satisfied with the Mirage zkSNARK [8].

Now, we will argue that $\mathcal{L}'_{\mathcal{I}}$ is sound (just as Lean's $\mathcal{L}_{\mathcal{I}}$ is) and that *C* is a sound randomized circuit for $\mathcal{L}'_{\mathcal{I}}$. Together, these facts imply that our zkSNARK is existentially sound for the language of true Lean theorems.

Type System Modifications. There are two differences between our type system and the Lean kernel's. First, our system's evaluation relation is the union of the kernel's evaluation and definitional equality relations (Sec. 4.2). Second, our system omits some constructs and judgements (i.e., quotient types, η -reduction, etc.). The omissions *shrink* the set of inhabited types, so they preserve soundness. However, our modified evaluation relation must be analyzed.

Our type system's soundness follows closely from prior work. Carneiro's analysis of Lean [52] distinguishes definitional equality as checked by the Lean kernel (which implements algorithms for term evaluation and equality checking) from an *ideal* definitional equality relation. The kernel cannot implement the ideal relation because it is undecidable. Carneiro shows that the kernel's equality relation is a subset of the ideal relation. Then, since the type system is sound when instantiated with the ideal relation, it is also sound for the kernel's relation. Our evaluation relation is also a subset of Carneiro's ideal equality relation (despite the difference in name), so our type system is sound by the same reasoning.

Circuit Soundness. Our randomized circuit *C* is *sound* (Sec. 2.1) for the language \mathcal{L}'_I . *C* checks a judgement tree in the type system that defines \mathcal{L}'_I , using randomness as an optimization. It uses randomness to reduce certain *polynomial* equalities (e.g., f(X, Y) =

g(X, Y)) to equalities over *evaluations* of those polynomials at a random input (e.g., $f(r_X, r_Y) = g(r_X, r_Y)$, for random $r_X, r_Y \in \mathbb{F}$).

This reduction is used by *C* in three ways. First, in ROM checking, for various ROMs. If ROM *i* has length n_i , values of length ℓ_i , and at most *K* accesses, then the soundness error here is $\leq 2\sum_i (1+\ell_i) (n_i+K)/|\mathbb{F}|$ (Sec. 2.1). Second, context entries are hashed to scalars using H_c with a random key. The probability of a collision between any pair of the N_C context entries is $\leq N_C^2/2|\mathbb{F}|$. Third, contexts are tested for (order-independent) equality using H_r with a random key. The number of inputs to each root-hash at most N_C , and the number of equalities is at most $\binom{N_C}{2}$. Thus, the probability of a collision in H_r for any equality test is at most $N_C^3/2|\mathbb{F}|$.

Thus, the total soundness error of C is at most

$$\frac{(4\sum_{i}(1+\ell_{i})(n_{i}+K))+N_{C}^{2}+N_{C}^{2}}{2|\mathbb{F}|}$$

In asymptotic terms, all of N_C , ℓ_i , n_i , K are poly(λ), and $|\mathbb{F}| \approx 2^{\lambda}$, so the soundness error is negligible in λ . Concretely, in our experiments (Sec. 7.3), N_C , n_i , K are at most 10^4 , ℓ_i is at most 30, and $|\mathbb{F}| > 2^{255}$. Thus, the soundness error of C is less than $2^{-214} \ll 2^{-128}$ (the target soundness of Mirage).

5.2 Knowledge Soundness and Zero-Knowledge

If one assumes Mirage is knowledge-sound, then zkPi's proof for CIC typing is also knowledge-sound. That is, for a \mathcal{P}^* to convince \mathcal{V} , it must know a CIC term that has type Thm2Type(ϕ). More precisely, there exists an efficient extractor \mathcal{E} that, for any Lean theorem ϕ and any efficient prover \mathcal{P}^* that convinces \mathcal{V} with all but negligible probability, $\mathcal{E}^{\mathcal{P}^*}(\phi)$: Thm2Type(ϕ) holds with all but negligible probability. We construct \mathcal{E} in Appendix E.1. Its correctness follows from the assumption that Mirage is knowledge-sound (App. C), the efficient invertibility of EncodeType, and the closure of the set of inhabited CIC types under zkPi's simplifier.

In an ideal zero-knowledge proof for Lean, only the theorem ϕ and the assumed axioms ax would be public. In zkPi, upper bounds on the number of judgements, contexts, and lifting judgements are public, as well as the inductive definitions ind used in the proof (these are all included in the instance). Prior work has similar bounds leakage [38], and it can be mitigated by artificially increasing the bounds, at the cost of proving time. Similarly, in practice many theorems are provable without private inductive declarations (e.g. all our example theorems, see Section 7.1), in which case the leakage of ind is immaterial. In Appendix E.2, we prove that zkPi is zero-knowledge. This follows directly from the fact that Mirage is zero-knowledge: the Mirage parameters encode only the aforementioned bounds and the Mirage instance encodes only (ϕ , ax, ind).

6 IMPLEMENTATION

To implement zkPi, we implement the sub-routines of Figure 5. That is, we implement: a term simplifier (\approx 2.4k lines of code, in Rust), an encoder (\approx 4k LOC, Rust), a circuit description (\approx 1k LOC, Z#), Mirage [8] (a 500 LOC patch to the bellman [53] library), and improvements to the CirC [7] Z# compiler (\approx 2.5k LOC, Rust). Our implementation is open-source.⁷

Simplifier. The simplifier attempts to simplify a CIC term t to an equivalent term t' of smaller *size* (i.e., fewer sub-terms). Smaller terms can be processed by a smaller type-checking circuit, which reduces the cost of creating the zkSNARK. For instance, for the and. comm theorem (Fig. 1a), Lean exports a term-type pair of size 154. Our simplifier reduces the size to 54. The simplifier also reduces the number of type-checking judgements needed. Without simplification, the and. comm example requires 479 judgements. The simplifier reduces this count to 300. Our simplifier is naive: it attempts to fully evaluate both the proof term and type. Sometimes, this may fail because even well-typed terms may take a long time to evaluate. Indeed, the Lean kernel itself often times-out when running the **#reduce** command over terms. We leave implementing a more intelligent simplifier as future work.

Parameters. Our circuit description depends on some parameters. That is, it describes a family of circuits indexed by these parameters. Essentially, the parameters encode the maximum size of of a proof checkable by that circuit, along various dimensions. Thus, proof creation requires a circuit with sufficiently big parameters.

The parameters comprise the maximum numbers of (non-lifting) judgements, lifting judgements, private terms, public terms, context nodes, and axioms used in type-checking. The theorem is encoded as public terms, while the proof is encoded with private ones. There are also four parameters for to inductive constructions: the number of declarations, the total number of constructors, the maximum number of recursive constructor arguments, and the maximum number of non-recursive constructor arguments.

When using zkPi, one must determine sufficiently large parameters for the application, generate a circuit, and run Setup. Then, one can create and verify proofs. Proving time and memory depend on circuit size, which depends on the parameter values.

7 EVALUATION

We evaluate in four ways. First, we prove example theorems in zero-knowledge. Second, we measure how many theorems from from Lean's standard library (stdlib) and from mathlib zkPi can prove with limited resources. These experiments show that despite limitations (Sec. 9), zkPi can handle real Lean proofs.

Third, we measure the dependence of zkPi's performance on system parameters (e.g., the number of judgements, terms, etc.). This shows the cost of different system components.

Lastly, we compare zkPi to ZKUNSAT [38], a prior work for interactive, non-succinct ZKPs for propositional unsatisfiability. This experiment shows that zkPi takes longer to prove statements about propositional unsatisfiability. However, we'll also see that the *generality* of zkPi can compensate for this.

7.1 Example Theorems

First, we prove some example theorems, which includes those mentioned in Section 1.1. See Appendix D for the full Lean source code of each example. We wrote the Lean proofs for these theorems, using alternate implementations of common standard library constructs (e.g. addition, subtraction, equality of natural numbers) to reduce term and proof size. We also state some basic facts (e.g., the transitivity of greater-than-or-equals) as public axioms (see Appendix D). Then, we compute suitable circuit size parameters cp

⁷The code is available at: https://anonymous.4open.science/r/zkpi-F31E/

Theorem	Prove (s)	Verify (s)	Setup (s)			# Constraints
			Simplify	Compile	Mirage.Setup	
and.comm	23.28	0.15	0.0021	37.83	25.24	538892
demorgan	85.10	0.26	1.30	152.89	74.34	1698306
degree_is_even	58.72	0.43	0.0037	102.82	56.02	1216795
pigeonhole	235.70	0.96	0.0097	455.19	208.64	4327799
dedup_list	1499.51	10.92	2.74	5086.72	1192.22	24384281
insertion_sort	2967.25	25.81	3.21	17480.79	2435.70	48615773

Figure 10: Proving time, verifying time, setup time, and constraint counts for zkPi as applied to the example theorems.

Library	Success (%)	Failure, by cause (%)					
		Memout	Timeout	Quotient Types	Inductive Families	Eta Expansion	Other
mathlib stdlib	14.1 57.9	54.5 18.5	8.1 16.9	20.5 2.2	0.3 1.1	0.4 1.5	2.1 1.9

Figure 11: The percentage of theorems from each library that we proved in zero-knowledge.

for each example. Finally, we run zkPi's Setup, Prove, and Verify algorithms to prove and verify the theorems in zero-knowledge. Figure 10 shows the proving time, verification time, setup time, and R1CS constraint count for each example. The proof size for all theorems is 240 bytes. We intentionally give examples with a wide range of proving times.

Setup is more expensive. However, typically one would run Setup once (with sufficiently large cp) and re-use the proving and verifying keys (pk, vk) to create and check *many* proofs. Thus, Setup's cost is amortized away.

7.2 Library Theorems

In this experiment, we run zkPi on many pre-proved Lean theorems. Our benchmark set is 2k theorems, half sampled from stdlib and half from mathlib. stdlib comprises \approx 10k theorems about basic objects: numbers, lists, sets, etc. [54]. mathlib is a community-developed Lean library of formal mathematics with \approx 100k theorems and over 1M lines of code [55, 56]. mathlib is broad: it spans analysis (e.g., the principle of analytic continuation), algebra (e.g., Hilbert's basis theorem), combinatorics (e.g., Hall's marriage theorem), and more. As before, we run Setup, Prove, and Verify for each benchmark. Our testbed is a GCP e2-highmem-8 machine, which has 8 cores and 64GB of memory. We limit each attempt to 30 minutes.

Figure 11 shows the percentage of theorems proven from each benchmark set, as well as the percentage which fail. We successfully prove and verify many theorems in zero-knowledge: 57.9% of the stdlib benchmark set and 14.1% of the mathlib benchmark set. There are two primary failure causes. First, some proofs are large enough that our pipeline exausts its time or memory limit. 54% of the mathlib benchmark set and 18% of the stdlib benchmark set fail because of memory exhaustion. Second, some proofs (especially in mathlib) use features that zkPi doesn't support. 23% of the mathlib benchmark set and 7% of the stdlib benchmark set fail because of unsupported features (quotient types, inductive families, η -expansion, etc.).

The reasons that many theorems exhaust the resource limits is because they have huge proofs. One key reason for this is because in our experiments we prove each theorem from first principles.



Figure 12: The number of theorems provable in various amounts of time.

Meanwhile, when Lean's kernel checks theorems, it checks each theorem assuming that previous theorems are correct. Incrementally proving theorems in zero-knowledge is a promising way of reducing resource costs. zkPi's support for axiom's already allows incremental proofs if the lemmas used by the theorem are public. We believe zkPi can be extended to do this privately using techniques like commit-and-prove or proof recursion. Another reason for large proof sizes is that many common Lean tactics like simp and rw can produce needlessly large proofs. For example, a proof that $\forall a, b \in \text{bool}, a || b = tt \rightarrow a = tt \lor b = tt$ using simp was 4060 judgements. A proof for the same theorem without simp was only 741 judgements. Looking at other ways to reduce proof sizes in Lean is another promising direction for future work.

Figure 12, shows the proving time for theorems proved from stdlib and mathlib. The maximum amount of time spent on any one proof is \approx 4.5; almost all of the rest of the pipeline's 30 minute time-limit is spent in Setup.

7.3 Performance Analysis

Now, we analyze how the size of the type-checking circuit depends on system parameters (Sec 6). Our size metric is the number of *rank-1 constraints* in the compiled circuit; this is essentially equivalent to the number of non-linear multiplications [57]. This metric is standard, because the time and memory costs of Setup and Prove both depend quasi-linearly on it. To measure the dependence on a parameter *p*, we set all other parameters to 1. We compile our circuit for $p \in \{10^i\}_{i=0}^4$ and measure constraint count. Then, multiple-linear

Parameter	Constraints
Judgements	1638
Lifting Judgements	282
Terms	44
Public Terms	132
Context Nodes	95
Axioms	10
Ind. Declarations	40
Ind. Constructors	94
Non-Rec Args	94
Rec Args	94

Figure 13: Total constraint count is a linear function of all parameters. This table shows the coefficients.



Figure 14: Proving time for the pigeonhole principle. ZKUN-SAT takes less time to produce a proof for a propositional representation (for fixed *n*). However, zkPi can prove pigeonhole directly (for all *n*) in a modest amount of time.

regression (which gives a near-perfect fit) shows the dependency of constraint count on each parameter. Figure 13 shows the results.

Judgements clearly has the largest impact: 1638 constraints each. There are three reasons for this. First, each judgement may require many additional RAM accesses. Because the circuit model does not allow for branching, these accesses are the union of the accesses required by all judgement rules. Second, checking a judgement entails additional judgement accesses (itself and two antecedents). Judgements are the largest Z# structure by far (they have 20 fields!), which must be "unpacked" into a sequence of fields at each access. Third, even without RAM, the judgement checking circuit is large because there are many different judgement types.

On the other hand, axioms, inductives, contexts, and terms are cheap because they add few or no RAM accesses, and they require few constraints to validate. Finally, the considerable difference between the cost of general judgements and lifting judgements justifies our decision to separate the them (Sec. 4.8).

7.4 Comparison to ZKUNSAT

ZKUNSAT [38] is an interactive, non-succinct ZKP for propositional unsatisfiability proved with resolution. Lean can also express resolution proofs [58], so we evaluate zkPi against ZKUNSAT. We use formulas that encode the pigeonhole principle (if n + 1 pigeons are placed in n holes, some hole has two pigeons) for each value of n [59]. We generate a resolution proof with PicoSAT [60], and use zkPi or ZKUNSAT to convert it to a zero-knowledge proof.

Figure 14 shows the time to refute the formula for different *n*. Each run is limited to 1 hour. For fixed *n*, ZKUNSAT (optimized for resolution) is faster than zkPi (which is general-purpose).

However, zkPi is able to prove more powerful statements than ZKUNSAT. For example, ZKUNSAT's resolution proofs can only show that pigeonhole holds for a specific value of n. Further, since resolution proofs of pigeonhole have size exponential in n [59], ZKUNSAT cannot handle large pigeonhole instances. In contrast, zkPi is able to prove the pigeonhole principle generically for *all* values of n in a single theorem in 235.7 seconds (see Section 7.1). More generally, since ZKUNSAT is for propositional proofs, it cannot directly prove any statement over an infinite domain (e.g. natural numbers, data types, Turing machine traces).

8 RELATED WORK

A long line of research proves correct computation evaluation in zero-knowledge [12], including proof systems for circuit evaluation and (bounded) RAM machine execution, and similar [7, 8, 13–35, 49, 57, 61–74]. We build on this research to implement a practical zkSNARK for Lean theorems. In addition to proving properties about a single program/circuit execution, a Lean theorem can show that a safety property holds for *all* executions.

In a closely related work, Luo et al. build a non-succinct, interactive ZKP of propositional unsatisfiability [38]. Meanwhile, we build a zkSNARK for a more expressive target (Lean theorems). A concurrent work extends this (interactive and non-succinct) approach to SMT theorems [75]. Some SMT proof calculi build on dependent typing [76–78], so zkPi's core could be used for SMT theorems too.

There are also interactive protocols to privately *solve* distributed satisfiability problems: including SAT [79] and computational tree logic [80]. These assume semi-honest behavior, so they do not constitute a ZK proof. In contrast, zkPi is succinct, non-interactive, and more expressive. Finally, Fang et al. build a ZKP for static program analysis [81].

9 DISCUSSION AND FUTURE WORK

Limitations and Future Work. zkPi's limitations suggest directions for future work. First, zkPi reveals the inductive declarations used in the proof (Sec. 5.2). Often this leakage is immaterial, because the inductive types are already public. For example, none of the proofs of our example theorems use private inductive types. In principle, one could remove this limitation with an in-circuit test that a declaration is well-formed.

Second, zkPi leaks upper bounds on the number of judgements, lifting judgements, and context entries (essentially, a 3D bound on proof size). This creates a mild trade-off between privacy and efficiency. Prior work has a similar tradeoff [38].

Third, zkPi does not support all of Lean: it does not handle quotient types, eta-expansion, universe parameterization (Sec. 4.5), nor all inductive declarations (Sec. 4.4). Many mathlib theorems use some of these features, so supporting them would increase the completeness of zkPi.

Conclusion. We have designed and constructed the first zkSNARK for Lean theorems. Our implementation works for a significant fraction of real Lean theorems from stdlib and mathlib. Our key technical contribution is a collection of techniques for zkSNARKs of dependent typing. One future application is distributing proprietary binaries with ZK proofs of their safety or correctness.

Acknowledgements. redacted

REFERENCES

- [1] Yves Bertot and Pierre Castéran. 2013. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer.
- [2] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In International Conference on Automated Deduction. Springer, 378–388.
- [3] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. Webpage: http://www.cis.upenn.edu/bcpierce/sf/current/index.html (2010).
- [4] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The Hunting of the SNARK. *Journal* of Cryptology 30, 4 (Oct. 2017), 989–1066. https://doi.org/10.1007/s00145-016-9241-9
- [5] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods in System Design* (2013).
- [6] William A Howard. 1980. The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism 44 (1980), 479–490.
- [7] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler infrastructure for proof systems, software verification, and more.
- [8] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. 2020. MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs. In USENIX Security. USENIX Association, 2129–2146.
- [9] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *ML*.
- [10] Lior Goldberg, Shahar Papini, and Michael Riabzev. 2021. Cairo a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063. https://eprint.iacr.org/2021/1063.
- [11] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. 1990. Everything Provable is Provable in Zero-Knowledge. In CRYPTO (LNCS, Vol. 403), Shafi Goldwasser (Ed.). Springer, Heidelberg, 37–56. https://doi.org/10.1007/0-387-34799-2_4
- [12] Michael Walfish and Andrew J. Blumberg. 2015. Verifying computations without reexecuting them: from theoretical possibility to near practicality. CACM (Feb. 2015).
- [13] Srinath T. V. Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. 2012. Making argument systems for outsourced computation practical (sometimes). In NDSS. The Internet Society.
- [14] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. 2012. Taking Proof-Based Verified Computation a Few Steps Closer to Practicality. In USENIX Security, Tadayoshi Kohno (Ed.). USENIX Association, 253–268.
- [15] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *IEEE S&P*. IEEE Computer Society Press, 238–252. https://doi.org/10.1109/SP.2013.47
- [16] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. 2013. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*.
- [17] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. 2014. Verifiable computation using multiple provers. Cryptology ePrint Archive, Report 2014/846. https://eprint.iacr.org/2014/846.
- [18] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In EUROCRYPT (LNCS, Vol. 9666), Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11
- [19] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE S&P.* IEEE Computer Society Press, 315–334. https://doi.org/10.1109/SP.2018.00020
- [20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In EUROCRYPT (LNCS), Vincent Rijmen and Yuval Ishai (Eds.). Springer, Heidelberg, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26
- [21] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. https://eprint.iacr.org/2019/953.
- [22] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Halo: Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Report 2019/1021. https://eprint.iacr.org/2019/1021.
- [23] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In ACM CCS. ACM Press, 2986–3001. https://doi.org/10.1145/3460120.3484556
- [24] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean

and Arithmetic Circuits. In 2021 IEEE S&P. IEEE Computer Society Press, 1074–1091. https://doi.org/10.1109/SP40001.2021.00056

- [25] Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2022. Orion: Zero Knowledge Proof with Linear Prover Time (LNCS). Springer, Heidelberg, 299–328. https://doi.org/10.1007/978-3-031-15985-5_11
- [26] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In CRYPTO (LNCS, Vol. 8043), Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6
- [27] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. 2013. A Hybrid Architecture for Interactive Verifiable Computation. In *IEEE S&P.* IEEE Computer Society Press, 223–237. https://doi.org/10.1109/SP.2013.48
- [28] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. 2013. Verifying computations with state. In SOSP. Extended version: http://eprint.iacr.org/2013/356.
- [29] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In USENIX Security, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 781–796.
- [30] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In CRYPTO (LNCS, Vol. 8617), Juan A. Garay and Rosario Gennaro (Eds.). Springer, Heidelberg, 276–294. https://doi.org/10.1007/978-3-662-44381-1_16
- [31] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In NDSS. The Internet Society.
- [32] Matthew Fredrikson and Benjamin Livshits. 2014. ZØ: An Optimizing Distributing Zero-Knowledge Compiler. In USENIX Security, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 909–924.
- [33] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile Verifiable Computation. In *IEEE S&P*. IEEE Computer Society Press, 253–270. https://doi.org/10.1109/SP.2015.23
- [34] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster Verifiable RAM with Program-Independent Preprocessing. In *IEEE S&P*. IEEE Computer Society Press, 908–925. https://doi.org/10.1109/SP.2018.00013
- [35] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJsnark: A Framework for Efficient Verifiable Computation. In *IEEE S&P*. IEEE Computer Society Press, 944–961. https://doi.org/10.1109/SP.2018.00018
- [36] Jacob Eberhardt and Stefan Tai. 2018. ZoKrates-Scalable Privacy-Preserving Off-Chain Computations. In IEEE Blockchain.
- [37] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler infrastructure for proof systems, software verification, and more. IEEE Computer Society Press, 2248–2266. https://doi.org/10.1109/SP46214.2022.9833782
- [38] Ning Luo, Timos Antonopoulos, William Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. 2022. Proving UNSAT in Zero Knowledge.
- https://eprint.iacr.org/2022/206. To appear in ACM CCS.
 [39] Nathan Wetzler, Marija JH Heule, and Warren A Hunt. 2014. DRAT-trim:
- Efficient checking and trimming using expressive clausal proofs. In SAT. [40] xenaproject. 2018. Learning Lean by example. https://xenaproject.wordpress.com/2018/12/30/learning-lean-by-example/.
- [41] Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl, and Jannis Limperg. 2021. The Hitchhiker's Guide to Logical Verification.
- [42] Srinath Setty, Justin Thaler, and Riad Wahby. 2023. Customizable constraint systems for succinct arguments. https://ia.cr/2023/552 https://ia.cr/2023/552.
- [43] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In EUROCRYPT (LNCS, Vol. 7881), Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Heidelberg, 626–645. https://doi.org/10.1007/978-3-642-38348-9_37
- [44] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable Zero Knowledge with No Trusted Setup. In *CRYPTO (LNCS, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Heidelberg, 701–732. https://doi.org/10.1007/978-3-030-26954-8_23
- [45] László Babai. 1985. Trading Group Theory for Randomness. In ACM STOC. ACM Press, 421–429. https://doi.org/10.1145/22145.22192
- [46] Abraham Waksman. 1968. A permutation network. JACM 15, 1 (1968).
- [47] C. Andrew Neff. 2001. A Verifiable Secret Shuffle and Its Application to e-Voting. In ACM CCS, Michael K. Reiter and Pierangela Samarati (Eds.). ACM Press, 116–125. https://doi.org/10.1145/501983.502000
- [48] Dan Boneh and Victor Shoup. 2021. A graduate course in applied cryptography. Draft 0.6.
- [49] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. 2013. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In *ITCS*, Robert D. Kleinberg (Ed.). ACM, 401–414. https://doi.org/10.1145/2422436.2422481
- [50] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. 2018. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program

Execution. In ASIACRYPT (LNCS, Vol. 11272), Thomas Peyrin and Steven Galbraith (Eds.). Springer, Heidelberg, 595–626. https://doi.org/10.1007/978-3-030-03326-2_20

- [51] Frank Pfenning and Christine Paulin-Mohring. 1989. Inductively defined types in the Calculus of Constructions. In International Conference on Mathematical Foundations of Programming Semantics.
- [52] Mario Carneiro. 2019. *The Type Theory of Lean*. Technical Report. Version 1.0.
 [53] Zcash developers. [n. d.]. Bellman Circuit Library and zkSNARK.
- https://github.com/zkcrypto/bellman. [54] The Lean Community. [n. d.]. Lean.
- https://github.com/leanprover-community/lean. Version 3.43.0.
 [55] The mathlib Community. 2020. The lean mathematical library. In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM. https://doi.org/10.1145/3372885.3373824
- [56] The mathlib Community. [n. d.]. mathlib. https://github.com/leanprover-community/mathlib.
- [57] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling Verifiable Computation Using Efficient Set Accumulators. In USENIX Security. USENIX Association, 2075–2092.
- [58] Tomaz Gomes Mascarenhas. 2023. Lean resolution proofs. https://github.com/ufmg-smite/leansmt/blob/main/Smt/Reconstruction/Certifying/Resolution.lean.
- [59] Armin Haken. 1985. The intractability of resolution. *Theoretical computer science* 39 (1985).
- [60] Armin Biere. 2008. PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation (2008).
- [61] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In ACM CCS, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2111–2128. https://doi.org/10.1145/3319535.3339817
- [62] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKS Without Trusted Setup. In CRYPTO (LNCS), Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 704–737. https://doi.org/10.1007/978-3-030-56877-1 25
- [63] Srinath Setty and Jonathan Lee. 2020. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275. https://eprint.iacr.org/2020/1275.
- [64] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-Efficient zkSNARKs Without Trusted Setup. In *IEEE S&P*. IEEE Computer Society Press, 926–943. https://doi.org/10.1109/SP.2018.00060
- [65] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. 2021. Brakedown: Linear-time and post-quantum SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/1043. https://eprint.iacr.org/2021/1043.
- [66] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. 2012. Practical verified computation with streaming interactive proofs. In *ITCS*, Shafi Goldwasser (Ed.). ACM, 90–112. https://doi.org/10.1145/2090236.2090245
- [67] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. 2012. Verifiable Computation with Massively Parallel Interactive Proofs. In *HotCloud*.
- [68] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In CRYPTO (LNCS, Vol. 11694), Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Heidelberg, 733–764. https://doi.org/10.1007/978-3-030-26954-8_24
- [69] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. [n. d.]. EMP-toolkit: Efficient Multi Party computation toolkit. https://github.com/emp-toolkit.
- [70] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In ACM CCS, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1687–1700. https://doi.org/10.1145/3243734.3243823
- [71] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In EUROCRYPT (LNCS), Vincent Rijmen and Yuval Ishai (Eds.). Springer, Heidelberg, 569–598. https://doi.org/10.1007/978-3-030-45727-3_19
- [72] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In ACM CCS. ACM Press, 2055–2074. https://doi.org/10.1145/3372297.3417283
- [73] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. 2021. Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs. In 2021 IEEE S&P. IEEE Computer Society Press, 1538–1556. https://doi.org/10.1109/SP40001.2021.00089
- [74] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In ACM CCS, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2075–2092. https://doi.org/10.1145/3319535.3339820
- [75] Daniel Luick, John Kolesar, Timos Antonopoulos, William R. Harris, James Parker, Ruzica Piskac, Eran Tromer, Xiao Wang, and Ning Luo. 2023. ZKSMT: A VM for Proving SMT Theorems in Zero Knowledge. https://ia.cr/2023/1762.

$$\frac{C \vdash f: \Pi(i: \llbracket [2\alpha] \rrbracket). \mathsf{add} \llbracket [\alpha] \rrbracket \llbracket [\alpha] \rrbracket}{C \vdash f_{j-1}: \llbracket [2\alpha] \rrbracket} \frac{(C, i \mapsto f_{j-1}) \vdash \mathsf{add} \llbracket [\alpha] \rrbracket \amalg \llbracket [2\alpha] \rrbracket}{C \vdash f_{j-1}: \llbracket [2\alpha] \rrbracket}$$

Figure 15: The recursive step in inference for t_{α}

- [76] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods in System Design* (2013).
- [77] Liana Hadarean, Clark Barrett, Andrew Reynolds, Cesare Tinelli, and Morgan Deters. 2015. Fine grained SMT proofs for the theory of fixed-width bit-vectors. In LPAR.
- [78] Duckki Oe, Andrew Reynolds, and Aaron Stump. 2009. Fast and flexible proof checking for SMT. In SMT Workshop.
- [79] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. 2022. ppSAT: Towards Two-Party Private SAT Solving.
- https://eprint.iacr.org/2021/1584. To appear in USENIX Security.
 [80] Samuel Judson, Ning Luo, Timos Antonopoulos, and Ruzica Piskac. 2020.
- Privacy Preserving CTL Model Checking through Oblivious Graph Algorithms. In *WPES*.
- [81] Zhiyong Fang, David Darais, Joseph P Near, and Yupeng Zhang. 2021. Zero Knowledge Static Program Analysis. In CCS.
- [82] Oded Goldreich and Yair Oren. 1994. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology* 7, 1 (Dec. 1994), 1–32. https://doi.org/10.1007/BF00195207
- [83] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In ACM CCS, Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby (Eds.). ACM Press, 62–73. https://doi.org/10.1145/168588.168596
- [84] Ran Canetti and Marc Fischlin. 2001. Universally Composable Commitments. In CRYPTO (LNCS, Vol. 2139), Joe Kilian (Ed.). Springer, Heidelberg, 19–40. https://doi.org/10.1007/3-540-44647-8_2

A REDUNDANCY HAS ASYMPTOTIC COST

We give a family of CIC term pairs $\{(t_{\alpha}, \tau_{\alpha})\}_{\alpha \in \mathbb{N}}$ such that

- the typing relation R defined as $\emptyset \vdash t_{\alpha} : \tau_{\alpha}$ holds
- with subcontexts, the inference DAG for *R* has size $\Theta(\alpha)$
- without, the inference DAG for *R* has size $\Theta(\alpha^2)$

Let nat be an inductive type representing the natural numbers through Peano arithmetic. For (metatheoretic) natural number $\alpha \in \mathbb{N}$, let $[[\alpha]]$ denote the inductive value for α . Let add : nat \rightarrow nat \rightarrow nat \rightarrow nat be the addition function for inductive naturals, defined recursively in its first argument.

Note that for all $\alpha, \beta \in \mathbb{N}$, for all contexts *C*, a reduction DAG of size $\Theta(i)$ shows that $C \vdash \operatorname{add} [[\alpha]] [[\beta]] \Downarrow [[\alpha + \beta]]$.

The type τ_{α} is

 $\Pi(a : [[2\alpha]]). \Pi(f : \Pi(i : [[2\alpha]]). add [[\alpha]] [[\alpha]]). [[2\alpha]]$

and term t_{α} is a function of *a* and *f* that applies *f* to *a* α times:

$$\lambda a. \lambda f. (f (f \cdots (f a) \cdots))$$

Proof size without subcontexts. We show that without subcontexts, the size of the inference DAG for *R* is $\Theta(\alpha^2)$. Let

 $f_0 \triangleq a$ $f_j \triangleq (f f_{j-1}) \qquad (j = 1, ..., \alpha)$ $F \triangleq \Pi(i : [[2\alpha]]). \text{ add } [[\alpha]] [[\alpha]]$

The inference DAG begins with lambda typing judgements

. . .

The inference continues recursively with application judgements. Let *j* be in $\{1, ..., \alpha\}$ and define $C \triangleq (a \mapsto [[2\alpha]], f \mapsto F)$. The inference $C \vdash f_j : [[2\alpha]]$ holds from the inference $C \vdash f_{j-1} : [[2\alpha]]$, as shown in Figure 15. This inference recurses to a base case $(C \vdash a : [[2\alpha]])$ in a total of α recursive steps. The inference for the type of f_j relies of the reduction

$$(C, i \mapsto f_{j-1}) \vdash \mathsf{add} \llbracket \alpha \rrbracket \llbracket \alpha \rrbracket \Downarrow \llbracket \alpha \rrbracket \downarrow \llbracket 2\alpha \rrbracket$$
(2)

In each instance of (2), the contexts differ, so each instance is justified by a DAG disjoint from all others. Thus, the total size of the inference is $\alpha \cdot \Theta(\alpha) = \Theta(\alpha^2)$.

Proof size with subcontexts. In a system that permits subcontexts in reduction inferences, the DAG can be smaller. Each instance of the reduction (2) follows from

$$C \vdash \mathsf{add} \llbracket \alpha \rrbracket \llbracket \alpha \rrbracket \amalg \llbracket \alpha \rrbracket \downarrow \llbracket 2\alpha \rrbracket \tag{3}$$

The latter reduction has an inference DAG size of $\Theta(\alpha)$, as before, but it can be used for all instances of (2). Thus, the total inference size is now $\Theta(\alpha)$.

B LISTING OF JUDGEMENT RULES

Figure 16 shows a listing of the judgement rules implemented within the system. Our rules use a few base relations:

- *lookup*(*C*, *n*): get name *n* from context *C*,
- get_args (Section 4.4),
- *lookup_ind(i)*: accesses the *i*th inductive declaration,
- *lookup_ind_ctor(i, c)*: accesses the *c*th constructor for the *i*th inductive declaration, and
- *get_args* (Section 4.4)
- *imax*: impredicative maximum, which is defined as *imax(u, v)* = if v == 0 then 0 else max(u, v)

They also reference a few term fields:

- "kind": e.g., application, inductive constructor, etc.
- "ind": a pointer to an inductive declaration (see Section 4.4) if this term refers to one,
- "head": the leftmost application argument, checked by the head relation (see Section 4.4),
- "ind_ctor": a pointer to the actually constructor term, if this is a constructor
- "argc": the number of application arguments applied to the head, check by the argc relation (see Section 4.4),

; as well as inductive declaration fields: defined in Section 4.4:

- "type": the inductive type
- "any_elim": a boolean which specifies that the recursor may resolve to any Sort
- "rec_body": the recursor's body
- "num_params": number of type parameters for the type
- "non_recs": number of recursive parameters, by constructor

Evan Laufer, Alex Ozdemir, and Dan Boneh

$$\frac{lookup(C, n) = e}{C + Var n|e} \left(\frac{C + e \parallel e'}{C + \lambda x.e \parallel \lambda x.e'} \left(\frac{dval Lam}{dval Lam} - \frac{C + a \parallel A'}{C + \Pi x : AB \parallel \Pi x : A', C) + B \parallel B'}{C + \Pi x : AB \parallel \Pi x : A', C) + B \parallel B'} \left(\frac{dval P}{C + f \parallel f'} \left(\frac{dval P}{C + f \parallel f'} \right) \left(\frac{C + f \parallel A', b + C + e \parallel e \mid e'}{C + f \parallel g'} \left(\frac{dval P}{C + f \parallel g'} \right) \right) \left(\frac{C + f \parallel A', b + C + e \parallel e \mid e'}{C + f \parallel g'} \left(\frac{dval P}{C + f \parallel g'} \right) \right) \left(\frac{C + f \parallel A', b + C + e \parallel e \mid e'}{C + f \parallel g'} \left(\frac{dval P}{C + f \parallel g'} \right) \right) \left(\frac{C + f \parallel A', b + C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{C + e \parallel e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{C + e \parallel e \mid e'}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{dval P}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{dval P}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \left(\frac{dval P}{C + e \parallel g'} \right) \right) \left(\frac{dval P}{C + e \parallel g'} \left(\frac{dval P}{C + e \parallel g'} \right) \left(\frac{dval P}{C + e$$

Figure 16: Listing of judgement rules implemented by the system

```
theorem pigeonhole (l : nat_list) : (lt_eff (list_len l) (sum_list l)) \rightarrow (double_pigeon l) :=
nat_list.rec (sub_eff_zero_zero)
  (\lambda (l_hd : N) (l_tl : nat_list) (l_ih : lt_eff (list_len l_tl) (sum_list l_tl) \rightarrow double_pigeon l_tl)
  (h : lt_eff (list_len (nat_list.cons l_hd l_tl)) (sum_list (nat_list.cons l_hd l_tl))),
    (em_eff (lt_eff (list_len l_tl) (sum_list l_tl))).cases_on
    (\lambda (h_1 : lt_eff (list_len l_tl) (sum_list l_tl)),
    (or.intro_right (lt_eff nat.zero.succ l_hd) (l_ih h_1)))
    (\lambda (h_1 : -lt_eff (list_len l_tl) (sum_list l_tl)),
    (or.intro_left
    (double_pigeon l_tl)
    (lt_eff_succ_gt h h_1))))
```

1

Figure 17: Pigeonhole proof

- "num_non_recs": number of non-recursive parameters, by constructor
- "rec_argc": the total number recursor arguments

Most of these are standard for CIC. The only main additions are the modified rules for the typing and evaluation of inductive types. The typing rule Type-Ind-Rec depends on the relation Well-Formed-Rec, which checks that the type T is correct for the given inductive type (as described in Sec. 4.5).

Prop. Recall (Sec. 4.5), that Prop (equivalently, Sort 0) is a distinguished type in Lean. It has some special typing rules. First, in Lean, inhabitant of the same proposition are definitionally equal. Since we combine definitional equality with evaluation, (Eval-Proof-Irrel) is our analogue of this rule. Also, while generally a Pi Πx : *Sort i, Sort j* has type to *Sort imax*(*i*, *j*), Πx : *Sort u*, Prop has type Prop. Through *imax*(*i*, *j*), which is 0 when j is and max(*i*, *j*) otherwise, the (Type-Pi) rule captures both.

Context Subsetting. Additionally, in all rules, we implicitly allow each antecedent context to be a subset of the conclusion context. Different antecedent contexts can be different subsets of the conclusion context. We omit this from our explicit description of the rules for brevity.

C ZKSNARKS FOR MA AND MIRAGE

Prior work [8] introduces the syntax and a construction for a zk-SNARK for an \mathcal{MA} language. However, they stop just short of defining security. In this section, we review the syntax of such a proof system and then define security. We conjecture that their construction fulfills our security definition.

Let \mathcal{L} be a language in \mathcal{MA} , with a deterministic, polynomial verification algorithm C(x, w, r) such that

- if $x \in \mathcal{L}$, $\exists w$ such that $\Pr_r[C(x, w, r) = \top] = 1$, and
- if $x \notin \mathcal{L}, \forall w, \Pr_r[C(x, w, r) = \top] \le \epsilon_s \le 1/2$.

Here ϵ_s is the soundness error. With *n* independent repetitions, one can improve ϵ_s to $O(2^{-n})$. Define R_C as

$$R_C = \{(x, w) : \Pr_r[C(x, w, r) = \top] = 1\}$$

Let \mathbb{L} be a set of $\mathcal{M}\mathcal{A}$ languages. A non-interactive proof system for \mathbb{L} is three polynomial-time, randomized algorithms Π = (Setup, Prove, Verify) with the following syntax:

- Setup(1^λ, L ∈ L) → (pk, vk): Given the security parameter and a language L, generate a proving key pk and a verifying key vk.
- Prove(pk, x, w) → π: Given a valid instance-witness pair, construct a proof that x is in the language that pk was generated for.
- Verify(vk, x, π) → {⊥, ⊤}: Accept or reject a proof that x is the language L that vk was generated for.

Completeness. Π is complete if for all $\mathcal{L} \in \mathbb{L}$, for all (pk, vk) in the support of Setup, and for all $(x, w) \in R_C$,

 $\Pr[\operatorname{Verify}(vk, x, \operatorname{Prove}(pk, x, w))] \ge 1 - \operatorname{negl}(\lambda)$

where $\operatorname{negl}(\lambda)$ denotes a function $f(\lambda)$ that is $o(\lambda^{-c})$ for all $c \in \mathbb{N}$.

Knowledge soundness. II is knowledge-sound if there exists a polynomial-time, randomized algorithm \mathcal{E} (the "extractor") such that for any $\mathcal{L} \in \mathbb{L}$ and for any polynomial-time, randomized algorithm \mathcal{P}^* , the following is $\leq \text{negl}(\lambda)$:

$$\Pr\left[\begin{array}{cc} (\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^{\lambda},\mathcal{L}) \\ (x,\pi) \leftarrow \mathcal{P}^*(\mathsf{pk}) & \wedge & w \leftarrow \mathcal{E}^{\mathcal{P}^*}(\mathsf{pk},x) \\ \mathsf{Verify}(\mathsf{vk},x,\pi) = \top & (x,w) \notin R_C \end{array}\right]$$

Zero-knowledge. Π is *zero-knowledge* if there exists a polynomialtime, randomized algorithm S (the "simulator") such that for all $\mathcal{L} \in \mathbb{L}$, for all (pk, vk) in the support of Setup, and for all $(x, w) \in R_C$, the following distributions are computationally indistinguishable:

$$\{\operatorname{Prove}(\operatorname{pk}, x, w)\} \approx \{\mathcal{S}(\operatorname{tk}, x)\}$$

where tk is the simulation trapdoor from Setup.

Succinctness. Π is succinct if the length of π and the runtime of Verify are both bounded by poly($\lambda + |x| + \log |C|$), where |C| is the evaluation time of *C*.

A *zkSNARK* for \mathbb{L} is a zero-knowledge succinct non-interactive argument of knowledge: a Π that is complete, knowledge-sound, zero-knowledge, and succinct.

Since non-interactive zero-knowledge proofs are impossible in the standard model [82] for \mathcal{NP} languages (assuming $P \neq \mathcal{NP}$), they are also impossible for \mathcal{MA} languages. But, there are many constructions of non-interactive zero-knowledge for \mathcal{NP} (and zk-SNARKs for \mathcal{NP}) in the random-oracle model [83] and the commonreference-string model [84]. We conjecture that the Mirage proof

Evan Laufer, Alex Ozdemir, and Dan Boneh

system can be shown to be a zkSNARK in the generic group model, or in the algebraic group model under a discrete-logarithm assumption [8].

C.1 Application to NP languages

Let C'(x, w, r) be a randomized circuit and C(x, w) be a deterministic circuit that defines an \mathcal{NP} language \mathcal{L} . We say that C' is *complete* with respect to C (or \mathcal{L}) if for all x, w such that $C(x, w) = \top$, $\Pr_r[C'(x, w, r) = \top] \ge 1$ and it is *sound* if for all x, w such that $C(x, w) = \bot$, $\Pr_r[C'(x, w, r) = \top] \le \operatorname{negl}(\lambda)$. It's easy to see that given a C' that is sound and complete for C, instantiating Mirage with C' gives a zkSNARK for the \mathcal{NP} language \mathcal{L} .

We use Mirage in this way to build zkPi's zkSNARK for CIC typing (Sec. 4). The language \mathcal{L} is the set of inhabited CIC types. The circuit *C* is the non-randomized type-checker, and we build a optimized and randomized circuit *C'* that we argue is sound for *C* (Sec. 5).

D FULL EXAMPLE PROOFS

The Lean code for the pigeonhole proof is listed in Figure 17. The full Lean code for our examples (including our custom standard library definitions) is open-source: https://anonymous.4open.science/r/zkpi-F31E/lean-examples/.

E SECURITY

E.1 Knowledge Soundness

The following extractor shows that zkPi is knowledge-sound. It relies on two procedures, DecodeTerm and Term2Pf.

```
\begin{split} \mathcal{E}^{\mathcal{P}^*}(\mathrm{pk}, x &= (\phi, ax, ind)) \to \pi \\ \tau &\leftarrow \mathrm{Simplify}(\mathrm{Thm}\,\mathrm{To}\,\mathrm{Type}(\phi))) \\ x' &\leftarrow \mathrm{Encode}\,\mathrm{Type}(\tau, ax, ind) \\ w &\leftarrow \mathrm{Mirage}.\mathcal{E}^{\mathcal{P}^*}(pk, x') \\ t &\leftarrow \mathrm{Decode}\,\mathrm{Term}(w, x') \\ \rho &\leftarrow \mathrm{Term}\,\mathrm{2Pf}(t) \\ \mathbf{return} \ \rho \end{split}
```

DecodeTerm is a deterministic, polytime algorithm that is an inverse of EncodeTermAndType. More specifically, if $x \leftarrow$ EncodeType(τ , ax, ind) and $\Pr_r[C(x, w, r) = \top] = 1$, then DecodeTerm(w, x) guarantees that its output t satisfies $\emptyset \vdash t : \tau$ (given axioms ax and inductive declarations ind). EncodeType exists because in EncodeTermAndType's witness encoding creates a witness which includes an encoding of a typing judgement for τ . That judgement includes a pointer to a term t that the circuit shows has type τ . DecodeTerm simply decodes t.

Term2Pf is a deterministic polytime algorithm that is an inverse of Pf2Term. Given a CIC term *t*, it constructs a Lean proof ρ such that $t = Pf2Term(\rho)$, e.g., by constructing and explicit Lean term.

Now we analyze our extractor. We will show that when Mirage. $\mathcal{E}^{\mathcal{P}^*}$ succeeds, our extractor does too. When the former succeeds, we have r, C(x, w, r) = 1, but then, per the property of DecodeTerm, we have that $\emptyset \vdash t : \tau$ (given axioms ax and inductive declarations ind). Then we have that $\emptyset \vdash \text{Term2Pf}(\rho) : \text{Thm2Type}(\phi)$, which, per the Curry-Howard correspondence, implies that ρ is a proof of ϕ .

One subtlety: in the foregoing proof, the typing relation ":" is the typing relation for *our* modification of Lean's type system. Thus, the Curry-Howard correspondence is being applied between Lean proofs and our type system. It is therefor important that the correspondence holds between Lean's proof language and it's type system, but between that proof language and our type system (as we argued in in Section 5).

E.2 Zero-Knowledge

Zero-knowledge is more straightforward. We give the simulator below. The inputs to the Mirage simulator are distributed exactly the same as the inputs to the Mirage verifier in the real protocol. Therefore, our simulators output is distributed exactly as the proof in the real protocol.

Simulate(tk, $x = (\phi, ax, ind)) \rightarrow \pi$ $\tau \leftarrow \text{Simplify}(\text{ThmToType}(\phi)))$ $x' \leftarrow \text{EncodeType}(\tau, ax, ind)$ **return** Mirage.Simulate(tk, x')