Less is more: refinement proofs for probabilistic proofs (extended version)*

Kunming Jiang,† Devora Chait-Roth, Zachary DeStefano, Michael Walfish, and Thomas Wies

NYU Department of Computer Science, Courant Institute †Now at Carnegie Mellon

Abstract. There has been intense interest over the last decade in implementations of *probabilistic proofs* (IPs, SNARKs, PCPs, and so on): protocols in which an untrusted party proves to a verifier that a given computation was executed properly, possibly in zero knowledge. Nevertheless, implementations still do not scale beyond small computations. A central source of overhead is the *front-end*: translating from the abstract computation to a set of equivalent arithmetic constraints. This paper introduces a general-purpose framework, called Distiller, in which a user translates to constraints not the original computation but an abstracted *specification* of it. Distiller is the first in this area to perform such transformations in a way that is provably safe. Furthermore, by taking the idea of "encode a check in the constraints" to its literal logical extreme, Distiller exposes many new opportunities for constraint reduction, resulting in cost reductions for benchmark computations of 1.3–50×, and in some cases, better asymptotics.

1 Introduction

Probabilistic proofs [7–9, 43–45]—PCPs, IPs, NIZKs, SNARKs, SNARGs, and so on—are fundamental in complexity theory and cryptography. They enable an untrusted *prover* to convince a *verifier* of some statement (for example, that a given computation Ψ, on specific input x, produces an alleged output y). In these protocols, the verifier does not inspect a classical witness to the truth of the statement (or re-execute Ψ) but instead checks an encoded proof *probabilistically*. Zero-knowledge variants allow the prover to keep some of the input to the computation—and the proof itself—hidden from the verifier. Astonishingly, the verifier's checks are (in some protocols) constant-time, regardless of the size of the computation [7, 8, 42]. The appeal of these properties in emerging application areas (most notably, outsourced computation, blockchains, and their intersection) has fueled intense interest in implementations over the last 13 years. The results have included 20 orders of magnitude reduction in costs, deployment of SNARKs in cryptocurrencies [35, 60, 79, 97], and an explosion of frameworks [88, 91, 94].

Yet, probabilistic proofs are heavily limited in scalability, making them impractical for general-purpose use (the hype notwithstanding). One source of costs is the *back-end*, which is the complexity-theoretic and cryptographic proving machinery. The other source of costs is the *front-end*, which translates high-level computations into the format that the back-end works over. In most probabilistic proof implementations, that format is some variant of *arithmetic constraints*: equations over a finite field.

Unfortunately, not only must the prover perform cryptographic operations proportional to the number of constraints (often with memory requirements that scale similarly), but also constraints are a verbose way to represent computations (§2). For example, every iteration of a loop requires separate constraints—likewise with all branches of conditional statements. Inequality tests, when translated into constraints, are expensive. So is RAM.

The question that we ask and answer in this paper is: *if back-end costs are here to stay and we are stuck translating computations to constraints, what can we do to mitigate costs?* Any such technique should achieve:

- *Conciseness.* Compared to a naive translation of a computation Ψ, we want to produce a smaller set of constraints.
- *Coupling.* There should be a way for the prover to actually satisfy the alternate constraints, which is non-trivial, since they may not correspond to the individual program steps that the prover takes to execute Ψ.

These two requirements have been addressed, at least partially. The authors of almost all front-ends observe that translation from a high-level computation Ψ need not result in constraints that simulate execution [26, 27, 30, 31, 50, 61, 70, 74, 76, 81, 83, 89, 92, 105] (§2). Rather, it suffices if the constraints are satisfiable iff the execution is valid. For example, consider a computation that invokes a quicksort subroutine. The naive approach is to compile quicksort into constraints. As an alternative [50, Appx. C], the prover can sort "outside the constraints", with the constraints enforcing that (a) the output is a permutation of the input [20, 93], and (b) this permutation is sorted. The naive approach requires $O(n \log n)$ inequality tests while the alternative requires only $O(n)$ inequality tests (for adjacent elements in part (b)). As inequality tests dominate this computation, the improvement is substantial.

We call such a checker of required properties a *widget*. This generalizes "gadget" [61], which refers to constraints that have been written by hand; widgets can additionally be encoded in a higher-level language and then compiled to constraints. Widgets have been proposed for arithmetic and bitwise operations [81], multi-precision operations [50], storage [27, 69], concurrent access to state [83], cryptographic operations [15, 25, 27, 31, 60, 69, 79], recursive composition [25, 31, 52], and optimization problems [5].

Yet, to the extent that these works make arguments about the correctness of substituting a computation with a widget, none of them provides formal justification: it is entirely possible that there are wrong widgets out there! Note that any such bug, even in an application that satisfies the other two requirements, destroys

*This version extends a conference publication at IEEE Security & Privacy, 2023. Compared to the conference publication, this version includes three appendices, improves and corrects several paragraphs in Section 5.5, and treats very recent related work in Section 7.

soundness of the end-to-end application. Thus, we add a third requirement:

- *Correctness.* This is not about correctness of the translation to constraints, which is crucial and complementary, and has been studied [38]. Our focus on correctness in this paper is on substitutions (of computations by a widget) that happen "upstream" of compilation-to-constraints.

There is work addressing this third requirement [89] but at the expense of the first two (§7).

This paper contributes a framework, *Distiller*, that addresses all three of these desiderata (§4). Distiller takes the goals in reverse order: it *starts* with Correctness. For each class of computations, the user writes down a specification of the computation, and proves a formal relationship between the implementation and the specification. This justifies compiling the specification (rather than the implementation) to constraints. This relationship is ensured by representing both the implementation and the specification as *transition systems* and adapting ideas from the theory of *refinement* [32, 55, 56, 62, 98]. A refinement relates the externally observable behaviors of two transition systems, formalizing the notion of correct substitution. A proof of refinement then yields a blueprint for the prover to satisfy the abstract constraints (Coupling). For Conciseness, a consequence of Distiller and its generality is to expose new opportunities for constraint reduction: Distiller lets us take the idea of widgets to its literal logical extreme.

We apply Distiller to a series of examples (§5), including binary search, convex hull, maximal strongly connected components (MSC), and minimum spanning tree (MST). In particular, the solutions we obtain for the latter two problems may be of independent interest. Our widget for MSC bears some resemblance to the checker for Tarjan's algorithm [86] proposed in prior work [24]. However, our solution builds on Dijkstra's MSC algorithm [33] and is specifically designed to obtain an efficient representation in constraints. For MST, we introduce the idea of encoding operations on an amortized data structure as a kind of special-purpose memory; our widget exploits this encoding to check the execution of Kruskal's algorithm [53] in a way that avoids the overhead of translating certain general-purpose computing structures (conditionals, loads, stores, loops with dynamic bounds) to constraints.

We implement and evaluate Distiller (§6). The system takes as input a program representing the implementation and specification transition systems, and generates outputs using two components. The first component partially automates the generation of refinement proofs relating the input transition systems. This component relies on additional user input in the form of proof annotations. The proofs can then be checked using the program verifier Viper [68]. Successful verification guarantees Correctness. The verification of our examples unveiled bugs that would have compromised Correctness in initial versions of two widgets. Once Correctness has been established, the second component enables Coupling by building on the Pequin toolchain [76]. This component produces two C programs that are provided as input to Pequin. One program expresses the part of the widget to be translated to constraints. The other program expresses the part to be computed outside the constraints. Pequin then translates the first program, executes the second program, and uses the obtained outputs to drive a probabilistic proof backend.

Finally, Conciseness: replacing an implementation by its specification does not guarantee more concise constraints. However, as we explain (§4–§5), we can often use Distiller to find, and establish the correctness of, an intermediate point between specification and implementation that does yield a substantial improvement (our work on MSC and MST, mentioned above, are examples of this). Concretely, in our examples, Distiller achieves reductions in constraint size ranging from small constant factors to asymptotic improvements for some problems, which for small problem instances already result in double-digit factors. Qualitatively, the more complex a computation, the more improvement Distiller generally yields. Computations with many memory accesses or searches of memory see particular benefit under Distiller.

Distiller is not perfect (§8). As a built system, its trusted computing base includes Pequin, Viper, and our own new translation front-end. However, this restriction is not fundamental.

The bottom line is that Distiller has taken a crucial step in improving front-ends: it has exhibited the logically most general way to exploit nondeterminism in arithmetic constraints, while doing so soundly, with performance improvements that range from good constants to orders of magnitude.

2 Background: applied probabilistic proofs

This section is intended to give just enough context for the rest of the paper. For a full, rigorous treatment of probabilistic proof implementations, see Thaler [88].

Back-end. In these setups, a *back-end* is a cryptographic or complexity-theoretic protocol between an untrusted prover $\mathcal{P}$ and a verifier $\mathcal{V}$ in which $\mathcal{P}$ convinces $\mathcal{V}$ that a given set of equations $\mathcal{C}$ has a solution.

In more detail, $\mathcal{V}$ and $\mathcal{P}$ (which are possibly probabilistic) agree on $\mathcal{C}$, as defined by a protocol, or defined by a *user* who invokes $\mathcal{V}$ and $\mathcal{P}$. The variables in $\mathcal{C}$ are elements in a finite field, typically $\mathbb{F}_p$ (the integers mod p), where p is a large prime (128 bits or more). For many back-ends, $\mathcal{C}$ is required to be in R1CS format [17, 18, 42, 74, 82]. R1CS generalizes arithmetic circuits, which generalize Boolean circuits. We refer to such a set of equations as *constraints*.

$\mathcal{V}$ does not trust anything $\mathcal{P}$ says; $\mathcal{P}$ can follow an arbitrarily malicious strategy (though some protocols presume a computational bound on $\mathcal{P}$ and cryptographic hardness assumptions of one kind or another).

$\mathcal{P}$ wants to prove to $\mathcal{V}$ that $\mathcal{P}$ holds a solution, or *satisfying assignment*, z to $\mathcal{C}$—but $\mathcal{V}$ does not want to receive z, and $\mathcal{P}$ may wish to keep z hidden. Instead, $\mathcal{P}$ gives $\mathcal{V}$ a certificate, possibly revealed interactively, which $\mathcal{V}$ checks. The guarantees are:

- *Completeness*: If $\mathcal{C}$ is satisfiable, then a correct $\mathcal{P}$ makes $\mathcal{V}$ accept, always (regardless of random choices made by $\mathcal{P}$, $\mathcal{V}$, or by the user in an offline phase).
- *Soundness*: If $\mathcal{C}$ is not satisfiable, the probability that $\mathcal{V}$'s checks pass is negligible (the probability is over random

choices made by the verifier or by the user in an offline phase). Some applications require a more general property, *Proof of Knowledge* (PoK): if $\mathcal{P}$ does not have access to a satisfying z (even if $\mathcal{C}$ is satisfiable), then $\mathcal{V}$ accepts with negligible probability. Note that these properties hold regardless of $\mathcal{P}$'s strategy.

- *Zero knowledge*: $\mathcal{V}$ gets no information about z other than what can be deduced from the fact that $\mathcal{C}$ can be satisfied.

Examples of recent back-ends are [22, 28, 29, 40, 41, 51, 52, 58, 63, 80, 95, 96, 101, 102]. These trade off different properties, including the nature of the cryptographic assumptions, noninteractivity, whether there is an offline phase, whether that phase has to be repeated each time the structure of $\mathcal{C}$ changes, and so on. However, in all of these works, the costs have a major dependence on the number of constraints, $|\mathcal{C}|$, and thus all of these works will benefit from improvements to front-ends.

Pipeline. Posit a user who cares about verifying the execution of some high-level computation Ψ, on some input x. $\mathcal{P}$ supplies y that is purportedly $\Psi(x)$, and wants to convince some $\mathcal{V}$, which is trusted by the user, that $y{=}\Psi(x)$. As a generalization, Ψ can be a relation, so the goal is to prove that $y \in \Psi(x)$. Existing implementations have the following pipeline:

Offline (one-time for Ψ):

0. The user writes down the computation Ψ.
1. The user compiles Ψ to constraints, $\mathcal{C}$, over variables X, Y, Z, where X and Y are vectors of variables that represent the inputs and outputs. This compilation needs to respect *Translation Fidelity*: for any x and y, $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable (by some $Z{=}z$) if and only if $y{=}\Psi(x)$ (or $y \in \Psi(x)$). Here, $\mathcal{C}(X{=}x, Y{=}y)$ means $\mathcal{C}$ with X bound to x ($\mathcal{V}$'s requested input) and Y bound to y (the purported output). As a small example, consider a computation that takes two inputs, computes their quotient (over a finite field, $\mathbb{F}_p$), and outputs that quotient plus 5. The corresponding constraints are: $\mathcal{C} = \{X_1 = Z_1 \cdot X_2, Y = Z_1 + 5\}$. Notice that for all pairs (x, y), $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable (by some $Z_1 = z_1$) iff $y = x_1/x_2 + 5$.
2. The user runs any setup procedure required by the back-end.

Online (for each x, y):

3. Given a specific input x, $\mathcal{P}$ identifies a satisfying assignment z to $\mathcal{C}(X{=}x, Y{=}y)$. In the simplest case, $\mathcal{P}$ does so by directly executing Ψ.[1]
4. $\mathcal{P}$ convinces $\mathcal{V}$ that it has, or knows, a satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$.

One property that we need from a pipeline is *End-to-end Completeness*: if $y = \Psi(x)$, then a correct $\mathcal{P}$ makes $\mathcal{V}$ accept with probability 1. This property relies on Translation Fidelity and (the back-end's) Completeness, together with the mechanics of Step 3. Another essential property is *End-to-end Soundness*: if $y \neq \Psi(x)$, then $\mathcal{V}$ rejects with overwhelming probability. This property relies on Translation Fidelity and (the back-end's) Soundness.

[1] Alternatively, $\mathcal{P}$ could possess auxiliary information that allows it to derive a satisfying assignment. A simple example is: Ψ requires $\mathcal{P}$ to supply the pre-image of a given CRHF H for a given digest, d. Then the input to Ψ is d, the output is M; $\mathcal{P}$ is then establishing that $M \in H^{-1}(d)$, but we do not think of $\mathcal{P}$ as "executing" H^{-1}; indeed, H^{-1} is presumed not to be efficiently computable.

Front-end. The front-end is Steps 1 and 3. We detail these steps below, incurring some textual debts to Buffet [92]. We focus on a compilation approach that we call the "ASIC approach". The alternative is the "CPU approach", which represents the execution of a CPU in constraints [15, 17–19, 104]. This results in much higher overhead [92].

Given a program, the compiler unrolls loops (each iteration gets its own variables), and converts the code to an intermediate form, for example static single assignment. The compiler then translates each line into one or more constraints [26, 27, 30, 31, 54, 70, 74, 76, 81, 92, 105]. Arithmetic and logical operations are concise. For example, the line of code `z3 = z2 + z1` becomes $\{Z_3 = Z_2 + Z_1\}$. By contrast, each inequality test and bitwise operation costs $\approx w$ constraints, where w is the bit width of the relevant variables (these operations work by separating a finite field element into bits [81, Appx.C]; see also [17, 74, 82, 92]). The combined set of constraints resulting from the line-by-line translation, and including RAM (see below), constitutes $\mathcal{C}$.

RAM operations (which we refer to as `LOAD` and `STORE` but which encompass any situation where an array index is not known at compile time) translate into variables that feed into a separate RAM-checking computation. This computation can take several forms. One is based on permutation networks and coherence checks [16, 18, 78, 92]. Loosely speaking, the computation (a) converts a time-ordered transcript of RAM operations into an address-ordered transcript of RAM operations with ties broken by execution order, and (b) uses pairwise checks in the address-ordered transcript to ensure that every `LOAD` delivers the value from the most recent `STORE`. Other techniques include Merkle trees and memory checking [15, 23, 27], polynomial identity testing [104], set accumulators [69], or even a brute force switch statement that considers every possible index (this works at small scales, as for some blockchain statements). Regardless of the representation, each `LOAD` and `STORE` is costly, as the RAM-checking computation has a number of constraints proportional to $\Omega(n \cdot r)$, where n is the number of operations, and r is the address width (log of memory size).

Solving. To produce a satisfying assignment, $\mathcal{P}$ in most pipelines (but not all [70]; see §7) goes constraint by constraint. The solution to some constraints is immediate; for example, given the constraint $Z_3 = Z_2 + Z_1$, if Z_1 and Z_2 are already determined then the setting to Z_3 is mechanically derived. Other constraints require nondeterministic input from the prover. Recall our earlier example: $\mathcal{C} = \{X_1 = Z_1 \cdot X_2, Y = Z_1 + 5\}$. Looking only at the constraint $X_1 = Z_1 \cdot X_2$, $\mathcal{P}$ knows X_1 and X_2 (they are inputs) but does not derive the setting of Z_1 by filling in other constraints. Rather, $\mathcal{P}$ computes X_2^{-1} "outside" the constraints (for example, using repeated squaring to compute X_2^{p-2}, which is X_2^{-1} in $\mathbb{F}_p$) and then sets Z_1 as $X_1 \cdot X_2^{-1}$. Other examples are inequality tests, where $\mathcal{P}$ supplies the values of each bit, and RAM-checking, where $\mathcal{P}$ supplies the settings for switches in a permutation network. In these cases, the process of translation from Ψ to constraints has to *decorate* certain constraints, to tell $\mathcal{P}$ how to solve them. (Decoration is known elsewhere as "annotation" [27, 70, 92], but later in this paper, we use "annotation" to mean something else.)

Widgets. Instead of representing certain operations directly

in constraints, one can sometimes substitute a validity check, as with the sorting example in the Introduction; we call this validity check a *widget*. The Pipeline handles such substitution. Assume for simplicity that only one operation in the computation Ψ has a widget, for example a single invocation of a sort() subroutine. Then Step 1 compiles the computation Ψ, but with the widget substituted for the direct operation. Meanwhile, Step 3 runs Ψ, with the direct operation. For this to work, the compiler must produce, and $\mathcal{P}$ must rely on, decorations. That is because $\mathcal{P}$ needs a way to connect the computation to the constraints, which no longer correspond to each other line-by-line.

When widgets enter the picture, achieving End-to-end Soundness and End-to-end Completeness requires an additional condition beyond the three that we have mentioned, namely Translation Fidelity, and (back-end) Completeness and Soundness. That additional condition is Correctness, from Section 1. Section 3 describes this condition informally; a precise definition requires machinery that we will build up in Section 4.

Costs and accounting. This paper's primary metric is $|\mathcal{C}|$. That is for two reasons. First, all back-ends in the literature impose costs on $\mathcal{P}$ (and, depending on the protocol, on $\mathcal{V}$) that are at least linear in the number of constraints, $|\mathcal{C}|$. Second, these costs typically dominate the cost to $\mathcal{P}$ of executing and solving (Step 3); thus, even though $\mathcal{P}$ executes the underlying computation, doing so contributes only negligibly to costs.

For concreteness, we sometimes assume the widely-used Groth16 backend [47, 61]. In Groth16, certificate size is constant (128 bytes) and $\mathcal{V}$ runs in constant time. However, the running time for $\mathcal{P}$ and for the setup phase are $O(|\mathcal{C}| \cdot \log|\mathcal{C}|)$. Because of this and memory bottlenecks from the access pattern, single-machine Groth16 provers are highly limited in the size of the computation that they can handle. There are works that take advantage of multiple machines [99] and heterogeneous hardware [103] to try to overcome these bottlenecks, but they too are limited. The bottom line is that *every* work in this research area will benefit from constraint sets with fewer constraints.

3 Motivating example: merging sorted lists

As noted in the introduction, an application of probabilistic proofs, at least in principle, is outsourcing computation. Those computations need not be "cryptographic". In fact, the mere act of outsourcing invites probabilistic proof machinery: a proof gives assurance that another entity executed correctly. Accordingly, our examples throughout this paper will have an algorithmic flavor, rather than employing cryptography. In particular, zero-knowledge guarantees provided by the back-end will be irrelevant. However, this is not fundamental, as zero-knowledge properties typically come for free in the back-end, and the Distiller framework applies just the same to cryptographic computations.

As an example algorithmic computation, consider `merge`, which takes as input multiple sorted lists with unique elements (unique across all lists) and outputs a sorted union of the elements. An example implementation of `merge`, which we denote T_I, is in Figure 1. When translated, `merge` comprises a number of constraints proportional to $L \cdot (\sum_k A_k.\texttt{len})$, because of the nested

```
1    void merge(L,A0,..,AL-1,B) {
2    ℓ0 : int[L] curr = {0};
3        int len, running_min, kstar; bool found;
4        len = 0;
5    ℓ1 : for (int k = 0; k < L; k++) {
6         len += Ak.len;
7        }
8        B.len = len;
9    ℓ2 : for (int i = 0; i < len; i++) {
10         found = false
11   ℓ3 :   for (int k = 0; k < L; k++) {
12             if (curr[k] < Ak.len && (!found ||
13                   Ak[curr[k]] < running_min)) {
14               running_min = Ak[curr[k]];
15               // running_min is the current min element
16               kstar = k;
17               // kstar indexes the list that contains
18               // running_min
19               found = true;
20               // indicates that branch has been taken
21             }
22         }
23         B[i] = running_min;
24         curr[kstar]++;
25       }
26   ℓ4 : return;
27   }
```

Figure 1: Pseudocode for the computation `merge`$(L, A_0, \ldots, A_{L-1}, B)$ (T_I). The precondition of `merge` requires that the $\texttt{A}_k$ are strictly sorted and their elements pairwise distinct. Also, there must be enough physical space in `B` to store the elements of all $\texttt{A}_k$.

loops on lines 9 and 11.

Observe that `merge` is *computing* its result. But in the setup of probabilistic proofs, the goal is to provide a proof about some alleged, exogenously-computed output. Thus, the set of constraints could instead *check* that a specification is met. We are interested in how to perform such a substitution systematically, meaning that the requirements in Section 1 are met.

A natural starting point is to translate the weakest logical specification (WLS) of `merge` that still expresses functional correctness: intuitively, one expects that logically weaker specifications "enforce less" and thus should yield smaller constraints when translated. Informally, the WLS is: "`merge`$(L, A_0, \ldots, A_{L-1}, B)$ terminates and, upon termination, B is monotonically increasing and holds just each element from $\{A_k\}$ exactly once." Pseudocode to check this specification, which we denote T_S, is depicted in Figure 2. Its complexity is $2 \cdot (\sum_k A_k.\texttt{len})$, which is an asymptotic improvement over $L \cdot (\sum_k A_k.\texttt{len})$ from earlier.

To read the pseudocode, note that the keyword **`havoc`** denotes a nondeterministic choice, while **`assume`** constrains choices. Concretely, when this pseudocode is compiled to $\mathcal{C}_{T_S}$ (§2), **`havoc`** statements become free variables that the prover supplies while **`assume`** statements become constraints that enforce the given statement. The specification uses **`for`**, which (logically) means bounded universal quantification, and (mechanically) unrolls and repeats the enclosed requirements.

In Figure 2, lines 4–12 constrain B to be sorted (in increasing

```
1  void merge_spec_naive(L,A_0,..,A_L-1,B) {
2    int k_i, j_i, i_kj;
3    havoc B.len;
4    for (int i = 0; i < B.len; i++) {
5      havoc B[i];
6      assume i == 0 || B[i-1] < B[i];
7      havoc k_i;
8      assume 0 <= k_i && k_i < L;
9      havoc j_i;
10     assume 0 <= j_i && j_i < A_k_i.len;
11     assume B[i] == A_k_i[j_i];
12   }
13   for (int k = 0; k < L; k++) {
14     for (int j = 0; j < A_k.len; j++) {
15       havoc i_kj;
16       // each element in some A_k is in B
17       assume 0 <= i_kj && i_kj < B.len;
18       assume A_k[j] == B[i_kj];
19     }
20   }
21   return;
22 }
```

Figure 2: Pseudocode for the weakest logical specification (T_S) of the `merge` computation. The precondition only requires that `B` has enough physical space for the elements of all A_k.

order), and enforce that $B \subseteq \bigcup_k A_k$. In particular, for each position i in B, the prover nondeterministically supplies *which list* (`k_i`) contributes to the ith position, and *which index* in that list (`j_i`) holds the contributed element. For the other direction, lines 13–20 specify that $\bigcup_k A_k \subseteq B$.

But how does the prover supply these values? Ideally they would result from simply executing the original computation.

This brings us to the Correctness and Coupling requirements (§1). We must prove a relationship between T_S and the actual code executed by the prover (T_I). The basic technique is to capture this relationship formally in terms of *refinement* [55, 56, 62]. A refinement proof coupling T_I and T_S not only establishes the correctness of the substitution, it also tells us how to augment T_I. The prover then executes the augmented implementation, which yields the values for the nondeterministically assigned variables in the specification.

A further improvement is possible. Notice that the implementation T_I (Fig. 1) uses the facts that the input lists are unique and sorted, whereas T_S (Fig. 2) uses neither fact. In the framework that we lay out in the sections ahead, we will have the freedom to choose a specification that refines the WLS yet still abstracts the computation. For example, by taking advantage of the uniqueness of the input lists, we obtain a less general but more concise specification than T_S. Specifically, we discard the lines in Figure 2 (13–20) that enforce $\bigcup_k A_k \subseteq B$, resulting in Figure 3, which we call T_E. When translated, T_E now yields a number of constraints proportional to $\sum_k A_k$.`len`, which saves a factor of two compared to T_S.

```
1  void merge_spec_efficient(L,A_0,..,A_L-1, B) {
2  ℓ'_0 : int k_i, j_i;
3  ℓ'_1 : havoc B.len;
4       assume B.len == Σ_{k=0}^{L-1} A_k.len;
5  ℓ'_2 : for (int i = 0; i < B.len; i++) {
6         havoc B[i];
7         assume i == 0 || B[i-1] < B[i];
8         havoc k_i;
9         assume 0 <= k_i && k_i < L;
10        havoc j_i;
11        assume 0 <= j_i && j_i < A_k_i.len;
12        assume B[i] == A_k_i[j_i];
13     }
14 ℓ'_4: return;
15 }
```

Figure 3: Pseudocode for the efficient specification (T_E) of the `merge` computation. The precondition is the same as for `merge` itself.

4 Framework

We formalize our framework in terms of *transition systems*, which provide a uniform formalism for representing both implementations and their specifications. From a semantic perspective, a transition system T defines a *language* $\mathcal{L}(T)$, which contains for each execution trace σ of T, a sequence of observations $o(\sigma)$ made about how T interacts with its environment during the execution. These observations may for instance encompass I/O, network traffic, etc.

We relate transition systems in terms of their languages. This allows us to formally capture when the execution of one transition system behaves like the execution of another, from the perspective of an external observer.

4.1 Transition systems and refinement

In our formalization, we adapt the classical setup of Abadi and Lamport [1]. A *transition system* $T = \langle \Sigma, \theta, \Delta, O, \alpha \rangle$ consists of a set of *states* Σ, a nonempty set of *initial states* $\theta \subseteq \Sigma$, a set of *transitions* $\Delta \subseteq \Sigma \times \Sigma$, a set of *observations* O, and an *observation function* $\alpha : \Sigma \to O$. Intuitively, the function α formalizes which aspects of a given state are observable. When T is known, we denote a transition $(s, s') \in \Delta$ by $s \to s'$ and say *s steps to s'*. We also call s' a *successor* of s.

Example 4.1. We illustrate with our motivating example (§3). We can regard T_I (Fig. 1) as defining a transition system $(\Sigma, \theta, \Delta, O, \alpha)$, as follows. The states Σ of T_I are mappings from program variables to values. For $s \in \Sigma$, we denote by $s.\mathtt{x}$ the value of program variable `x` in s. We sometimes write x for a value of the program variable `x` when the state s is unspecified. We write $s[\mathtt{x} \mapsto v]$ to denote the new state obtained from s by updating the value of `x` to v and keeping the values of all other program variables unchanged. The program variables include a dedicated variable `pc` storing the value of the program counter, which ranges over the control locations $\ell_0, \ldots, \ell_4$. (For simplicity of exposition, we are treating the execution of a basic block, such as one iteration of a non-nested loop, as a single transition.)

The observations O of T_I are the values of the input arrays and

output array at the program start and return. Intuitively, these are the values that an external user can observe from the program. All intermediate program states of the computations are unobservable, which we denote by the special observation τ. Formally, we define O using the following grammar:

$$O ::= \mathsf{in}(L, A_0, \ldots, A_{L-1}, B) \mid \mathsf{out}(L, A_0, \ldots, A_{L-1}, B) \mid \tau \ .$$

The observation function $\alpha : \Sigma \to O$ is then defined as follows:

$$\alpha(s) = \begin{cases} \mathsf{in}(s.\mathsf{L}, s.\mathsf{A}_0, \ldots, s.\mathsf{A}_{(s.\mathsf{L}-1)}, s.\mathsf{B}) & \text{if } s.\mathsf{pc} = \ell_0 \\ \mathsf{out}(s.\mathsf{L}, s.\mathsf{A}_0, \ldots, s.\mathsf{A}_{(s.\mathsf{L}-1)}, s.\mathsf{B}) & \text{if } s.\mathsf{pc} = \ell_4 \\ \tau & \text{otherwise} \ . \end{cases}$$

The transitions Δ of T_I are obtained from the program description in the expected way. For instance, the body of the for loop at control location ℓ_1 yields all transitions $s \to s'$ such that $s.\mathsf{pc} = \ell_1$, $s.\mathsf{k} < s.\mathsf{L}$, and

$$s' = s[\mathtt{len} \mapsto s.\mathtt{len} + s.\mathsf{A}_{(s.k)}.\mathtt{len}][\mathsf{k} \mapsto s.\mathsf{k} + 1] \ .$$

The set of initial states θ consists of all states s that satisfy the precondition of T_I (Figure 1). We assume that this precondition is specified by a formula φ_{pre}. That is, φ_{pre} states that $\mathsf{pc} = \ell_0$, and that the arrays A_k are sorted in strictly increasing order and its elements pairwise distinct. We write $s \models \varphi_{\mathsf{pre}}$ to indicate that s satisfies φ_{pre}.

An infinite sequence of states σ is called an *(execution) trace* of T if it starts in an initial state and respects T's transition relation: formally, $\sigma_0 \in \theta$ and for all $i \geq 0$, either σ_i steps to σ_{i+1} or $\sigma_i = \sigma_{i+1}$ and σ_i has no successors in Δ. If $\sigma_i = \sigma_{i+1}$, we say that σ *stutters* in step i. A terminating execution of T corresponds to a trace that stutters forever in its final state. By abuse of notation, we write $\alpha(\sigma)$ to denote the sequence of observations obtained by applying α pointwise to the states in σ. We denote the set of all traces of T by $\mathsf{traces}(T)$.

Let $\sharp$ be the function that maps a sequence σ to the sequence obtained from σ by replacing all repeated consecutive copies of elements by a single copy, for example, $\sharp(\langle 0,0,1,1,1,2,3,3,3,3\rangle) = \langle 0,1,2,3\rangle$.

The *language* of T, denoted $\mathcal{L}(T)$, is defined by applying α pointwise to each trace in $\mathsf{traces}(T)$ and then removing stutters. The intuition for removing stuttering is that we want to capture only the observable behavior: stuttering steps correspond to unobservable internal computation steps. Formally, we define the sequence of observations $o(\sigma)$ made from a trace σ as $o(\sigma) \stackrel{\text{def}}{=} \sharp(\alpha(\sigma))$ and then let

$$\mathcal{L}(T) \stackrel{\text{def}}{=} \{o(\sigma) \mid \sigma \in \mathsf{traces}(T)\} \ .$$

Example 4.2. In the motivating example (§3), the language of the transition system T_I is simply

$$\mathcal{L}(T_I) = \left\{ \langle \alpha(s), \tau, \alpha(s') \rangle \;\middle|\; s \models \varphi_{\mathsf{pre}} \wedge s' \models \varphi_{\mathsf{post}} \right\} \ .$$

Here, the precondition φ_{pre} is as defined above. The postcondition φ_{post} states that $\mathsf{pc} = \ell_4$, B is sorted in strictly increasing order, and the set of elements of B is equal to the union of the set of elements of the arrays A_k. The single τ in each observation sequence in $\mathcal{L}(T_I)$ summarizes all intermediate states of the computation.

A transition system T_I *refines* another transition system T_S iff $\mathcal{L}(T_I) \subseteq \mathcal{L}(T_S)$. This definition captures the idea that from the perspective of an external observer, every execution of T_I behaves like some execution of T_S. Typically, we think of T_S as *the specification* and T_I as *the implementation*. We denote a refinement relationship by $T_I \leq T_S$.

A classical approach to proving refinement relationships is to construct a *refinement mapping*. Formally, a refinement mapping between T_I and T_S is a function $r : \Sigma_I \to \Sigma_S$ such that

1. $r(\theta_I) \subseteq \theta_S$,
2. $\forall s \in \Sigma_I$, $\alpha_I(s) = \alpha_S(r(s))$, and
3. $\forall s, s' \in \Sigma_I$, if $s \to_I s'$, then $r(s) \to_S r(s')$ or $r(s) = r(s')$.

The first property states that r maps the initial states of T_I to those of T_S. The second property states that the observations computed from states are preserved by r. The third property states that every transition in Δ_I is matched by a corresponding transition in Δ_S under r or by a stuttering step. Together, these properties capture the intuition that the relationship between a refinement and its specification is that the specification abstracts steps that are "internal" to the implementation.

Once it has been established that $r : \Sigma_I \to \Sigma_S$ is a refinement mapping, $T_I \leq T_S$ follows: given a trace σ_I of T_I, the sequence $r(\sigma_I)$ is a trace of T_S (modulo stuttering). Moreover, $r(\sigma_I)$ makes the same observations as σ_I, i.e., $\sharp(\alpha_I(\sigma_I)) = \sharp(\alpha_S(r(\sigma_I)))$. Hence, the existence of r establishes that T_I refines T_S.

We write $T_I \leq_r T_S$ to indicate that r is a refinement mapping between T_I and T_S. An important property that we will use freely later is that refinement mappings compose: $T_1 \leq_r T_2$ and $T_2 \leq_q T_3$ implies $T_1 \leq_{q \circ r} T_3$.

4.2 Refinement-based widgets

We can now use the language of transition systems to recast Steps 0, 1, and 3 in Section 2 and explain how widgets are conventionally used to modify these steps. We start from a given transition system T_I and a property $\phi \subseteq O^\omega$ specifying the observation sequences of interest (Step 0). The problem is for the prover $\mathcal{P}$ to convince the verifier $\mathcal{V}$ that $\mathcal{L}(T_I) \cap \phi$ is nonempty. Here, the property ϕ will, in particular, ensure that the considered observations are restricted to those that are bound to the specific input x and alleged output y. However, ϕ may impose additional requirements on the observation sequences that are of interest to $\mathcal{V}$. The conventional approach is then to first translate T_I into constraints $\mathcal{C}_{T_I}(\phi) = \mathcal{C}_{T_I} \wedge \mathcal{C}_{o_I^{-1}(\phi)}$. We elide the definition of $\mathcal{C}_{o_I^{-1}(\phi)}$. In the context of the steps in Section 2, it is simply $X = x \wedge Y = y$. The translation guarantees that $\mathcal{C}_{T_I}(\phi)$ is satisfiable iff $o(\sigma_I) \in \mathcal{L}(T_I) \cap \phi$ for some σ_I (Step 1). The prover then executes T_I on the specified input x to obtain such a σ_I and derives from it the desired satisfying assignment (Step 3).

The conventional use of a widget is then to replace the constraints $\mathcal{C}_{T_I}$ by a smaller set of constraints $\mathcal{C}_{T_W}$. The prover still executes T_I to yield σ_I, but uses σ_I to compute a satisfying assignment for $\mathcal{C}_{T_W}(\phi)$. A crucial shortcoming of this approach is that replacing T_I by T_W is not formally justified. In particular, there is no guarantee that the existence of a satisfying assignment for

$\mathcal{C}_{T_W}(\phi)$ implies the nonemptiness of $\mathcal{L}(T_I) \cap \phi$, potentially compromising the soundness of the proof system. Moreover, there is no systematic approach to compute a satisfying assignment for $\mathcal{C}_{T_W}(\phi)$ from σ_I. We use the notion of refinement to address both of these shortcomings.

First, we change the problem setup as follows. The new Step 0 is to write down transition systems T_S, T_E, and T_I, as well as refinement mappings r and q such that $T_I \leq_r T_E \leq_q T_S$. This is our formal definition of Correctness (§1): a widget represented as T_E is Correct if it satisfies the refinement chain $T_I \leq_r T_E \leq_q T_S$. Now, the problem is for the prover $\mathcal{P}$ to convince the verifier $\mathcal{V}$ that $\mathcal{L}(T_S) \cap \phi$ is nonempty. That is, $\mathcal{V}$ is only interested in T_S, the weakest specification; the transition systems T_E and T_I are merely a means to an end to solve the problem. T_E then plays the role of T_W above. The new Step 1 is to translate T_E and ϕ into constraints $\mathcal{C}_{T_E}(\phi)$. The new Step 3 is for $\mathcal{P}$ to execute T_I on x (obtaining $o(\sigma_I) \in \mathcal{L}(T_I) \cap \phi$), to use r to compute a trace $r(\sigma_I)$, and finally to use $r(\sigma_I)$ to compute a satisfying assignment for $\mathcal{C}_{T_E}(\phi)$.

Observe that $T_E \leq_q T_S$ implies that if $o(\sigma_E) \in \mathcal{L}(T_E) \cap \phi$ for some σ_E, then $o(q(\sigma_E)) \in \mathcal{L}(T_S) \cap \phi$. Hence, assuming Translation Fidelity, if $\mathcal{C}_{T_E}(\phi)$ is satisfiable, then $\mathcal{L}(T_S) \cap \phi$ is nonempty. This ensures the soundness of the approach. Similarly, $T_I \leq_r T_E$ means that if $o(\sigma_I) \in \mathcal{L}(T_I) \cap \phi$, then $o(r(\sigma_I)) \in \mathcal{L}(T_E) \cap \phi$ and, hence, $\mathcal{C}_{T_E}(\phi)$ is satisfiable. This ensures the completeness of the approach.

A difference between proving $T_I \leq_r T_E$ and $T_E \leq_q T_S$ is that q need not be explicit. That is, although End-to-end Soundness requires that if $\mathcal{C}_{T_E}(\phi)$ is satisfiable then so is $\mathcal{C}_{T_S}(\phi)$, the actual satisfying assignment to $\mathcal{C}_{T_S}(\phi)$ is not used explicitly. Consequently, $T_E \leq T_S$ can be established by means other than refinement mappings, for example a proof based on simulation relations [65, 73, 90].

We note that the approach also applies in the special case where $T_S = T_E$. Though, generally, the crux is to find a suitable T_E *in between* T_S and T_I that yields a reduction in the constraint size relative to both T_S and T_I.

Constructing refinement mappings. It remains to show how to construct refinement mappings. We demonstrate this with the merge computation as a guiding example, using general principles inspired by refinement calculi such as [64, 66, 67]. These principles apply broadly (Section 5 contains further examples).

Our first step is to construct a refinement mapping r between the transition system T_I of the merge computation (Fig. 1) and its intermediate specification T_E (Fig. 3). As we will explain below, r can then be used to obtain a satisfying assignment for the constraints $\mathcal{C}_{T_E}$ from a given execution of T_I, enabling more efficient verification of that execution. In a second step, we then show that the intermediate specification T_E refines the naive specification T_S.

To prove $T_I \leq T_E$, we divide the construction of the refinement mapping into three steps by deriving two auxiliary transition systems T_{IE} and $\hat{T}_{IE}$ that yield a refinement sequence $T_I \leq T_{IE} \leq \hat{T}_{IE} \leq T_E$. Intuitively, the auxiliary transition systems couple T_I and T_E so that they are executed together.

A transition system T_E refined by an implementation T_I will typically involve nondeterministic (**havoc**, see §3) assignments to

```
1   void merge (L,A0,..,AL-1,B) {
2   ℓ0 : int[L] curr = {0};
3       int len, running_min, kstar, k_i, j_i;
4       bool found;
5       len = 0;
6   ℓ1 : for (int k = 0; k < L; k++) { len += Ak.len; }
7       B.len = len;
8       assume B.len == ∑_{k=0}^{L-1} Ak.len;
9   ℓ2 : for (int i = 0; i < B.len; i++) {
10        found = false;
11  ℓ3 :   for (int k = 0; k < L; k++) {
12          if (curr[k] < Ak.len && (!found ||
13                Ak[curr[k]] < running_min)) {
14            running_min = Ak[curr[k]];
15            kstar = k;
16            found = true;
17          }
18        }
19        B[i] = running_min;
20        assume i == 0 || B[i-1] < B[i];
21        k_i = kstar;
22        assume 0 <= k_i && k_i < L;
23        j_i = curr[kstar];
24        assume 0 <= j_i && j_i < Ak_i.len &&
25               B[i] == Ak_i[j_i];
26        curr[kstar]++;
27      }
28  ℓ4: return;
29  }
```

Figure 4: Pseudocode for the transition system $\hat{T}_{IE}$. The prover will execute the black and blue code (T_{IE}) instead of T_I. The values in blue are used to create an assignment to the nondeterministic variables occurring in the constraints obtained from T_E (the red **assume** statements).

program variables that do not appear in the implementation. In our example of the merge computation, these are the assignments to `k_i` and `j_i` in Figure 3 (Lines 8 and 10). The execution of T_E can proceed only if the value chosen by each nondeterministic assignment satisfies the constraints imposed by the subsequent **assume** statements. A key step in the refinement proof is therefore to show that such values can be obtained from the trace σ_I. We make this step explicit in the construction of the intermediate transition system T_{IE}. This transition system augments T_I with those variables unique to T_E as well as assignments to these variables that determine the desired values to be chosen for the nondeterministic assignments in T_E. In our example, this augmentation can be seen in lines 3, 21, and 23 shown in blue in Figure 4.

Observe that the assignments to `k_i` and `j_i` in T_{IE} of our example depend only on the original program variables of T_I. Moreover, the variables do not interfere with the other parts of the transition system in any way. Such auxiliary variables that are used only for the purpose of proving a refinement relation are sometimes referred to as *ghost variables*. Conveniently, if adding ghost variables to a transition system T results in T', then $T \leq T'$ [64]; thus $T_I \leq T_{IE}$.

In the context of program translation for probabilistic proofs, augmenting an implementation with ghost variables is not only useful for proving the refinement between T_I and T_E. The system

T_{IE} also instructs the prover how to obtain the satisfying assignment for the constraints $\mathcal{C}_{T_E}$. That is, the prover will actually execute T_{IE} instead of T_I.

The next step in our construction is to augment T_{IE} with the **assume** statements in T_E that constrain the values chosen for the nondeterministic assignments. We call the resulting transition system $\hat{T}_{IE}$. In our merge example, $\hat{T}_{IE}$ is shown in Figure 4 with the added **assume** statements highlighted in red (Lines 8, 20, 22, and 24).

Establishing the refinement $T_{IE} \leq \hat{T}_{IE}$ follows a generic construction. We first show that the added **assume** statements express invariants of T_{IE}. That is, the assumed expressions must always evaluate to `true` in T_{IE}, at the appropriate program points. In Appendix A, we discuss this step of the proof in more detail with regards to the merge computation. Once the invariants have been established, $T_{IE} \leq \hat{T}_{IE}$ follows, by simply using the identity function on the states of T_{IE} as the refinement mapping.

Finally, we observe that T_E can be obtained from $\hat{T}_{IE}$ by abstracting all program variables that appear in T_I but not in T_E. For our merge example, this amounts to removing the loops at locations ℓ_1 and ℓ_3 in $\hat{T}_{IE}$, and replacing the assignments on lines 7, 19, 21, and 23 that depend on the abstracted program variables by **havoc** commands.

Abstracting program variables in this systematic manner again yields a refinement by construction. The refinement mapping changes the value of the program counter in the expected way. For instance, the refinement mapping in our example coalesces locations ℓ_2 and ℓ_3 to ℓ'_3 and maps all other locations ℓ_i to ℓ'_i. The values of the remaining program variables that are common to $\hat{T}_{IE}$ and T_E are preserved by the refinement mapping. This concludes the proof of $T_I \leq T_E$.

It remains to argue that T_E refines T_S. One can generally apply the above technique again, to construct an appropriate refinement mapping. In particular, one can show that the following property is an invariant at the end of the for loop in T_E (Fig. 3):

```
∀ k,j :: 0 <= k && k < L && 0 <= j && j < Ak.len ==>
  ∃ i :: 0 <= i && i < B.len && Ak[j] == B[i]
```

The second for loop at lines 13 to 20 of T_S (Fig. 2) establishes exactly the same property.

Systems view. An end-to-end system view of Distiller is as follows. At compile-time the user provides a weakest specification T_S, an effective specification T_E, and a computation T_I (the new Step 0 in Section 2). One must then show that the refinement relationships $T_I \leq T_E$ and $T_E \leq T_S$ hold. These proofs can be done by the user outside of the system or the system aids the user by (partially) automating the proofs.

Such a refinement proof (say, $T_I \leq T_E$) can be constructed generically in the following way. First, one augments T_I with the necessary ghost variables (yielding T_{IE}) to obtain $T_I \leq T_{IE}$. Then one adds the invariants needed to properly constrain the nondeterministic assignments in T_E (yielding $\hat{T}_{IE}$) to obtain $T_{IE} \leq \hat{T}_{IE}$. To take the final step to T_E, one abstracts away all variables that are found in T_{IE} but not T_E to obtain $\hat{T}_{IE} \leq T_E$. One proceeds similarly for $T_E \leq T_S$.

$\hat{T}_{IE}$ is a *coupling* of T_I and T_E that makes explicit how the **havoc**ed ghost variables in T_E are computed from T_I. $\hat{T}_{IE}$ is then the input to an augmented front-end that splits $\hat{T}_{IE}$ into T_E and T_{IE}. It then compiles T_E to constraints $\mathcal{C}_{T_E}$ (the new Step 1). For each invocation of the probabilistic proof protocol (the new Steps 3 and 4), the prover runs T_{IE} and feeds its values back in to get a satisfying assignment to $\mathcal{C}_{T_E}$.

Note that $T_E \leq T_S$ is needed for End-to-end Soundness (every satisfying assignment to $\mathcal{C}_{T_E}$ encodes an element of $\mathcal{L}(T_S)$) while $T_I \leq T_E$ is needed for End-to-end Completeness (a satisfying assignment to $\mathcal{C}_{T_E}$ can be obtained from T_I).

Example		Improvement
Merging	(Ch. 16)	$\Theta(L)$
Find Min	(Ch. 12)	$1.4\times$
Binary Search	(Ch. 12)	$\Theta(\log(n)\log(\log(n)))$
Pattern Matching	(Ch. 18)	$3\times$
Next Permutation	(Ch. 13)	$1.4\times$
Dutch Flag	(Ch. 14)	$1.5\times$
RR Sequence	(Ch. 17)	$2\times$
Sum of Powers	(Ch. 19)	$1.66\times$
2D Convex Hull	(Ch. 24)	$5\times$
2D Convex Hull*	(Ch. 24)	$\Theta(\log(n))$
MSC	(Ch. 25)	$17.5\times$
MST	(Ch. 22)	$52.2\times$

Figure 5: Improvement for all examples based on theoretical analysis on large inputs. For improvements where T_E has asymptotically fewer constraints than T_I, we provide the complexity of the improvement; otherwise we provide a constant factor. L in Merging is the number of lists. n in Binary Search is the length of the array. n in 2D Convex Hull is the total number of nodes, and 2D Convex Hull* is the case where the nodes in the convex hull are marked instead of returned in a list.

5 Examples

We have applied the Distiller framework to the problems in Dijkstra's classic book *A Discipline of Programming* [33]. We chose this source for two reasons. First, it discusses algorithms for a diverse set of problems. Second, Dijkstra develops his algorithms iteratively, starting from a formal problem specification. This approach helps to identify suitable intermediate transition systems T_E that yield an efficient translation to constraints.

Our evaluation considers 11 of the 14 problems discussed in Dijkstra's book. The three problems we have omitted are "Updating a sequential file" (Chapter 15), "The problem of the smallest prime factor of a large number" (Chapter 20), and "The problem of the most isolated villages" (Chapter 21). We also have simplified the problem of computing the convex hull in three dimensions (Chapter 24) to the two-dimensional case.

For all the problems that we have considered, we are able to obtain significant reductions in the size of the generated constraints (Fig. 5). In some cases, the scale factor of the reduction grows asymptotically with the problem instance size.

In the following, we discuss a selected subset of the considered problems in detail. We explain T_S, T_E, and T_I for these problems, provide a qualitative analysis that explains the expected reduc-

```
1   int find_min(n, A, B) {
2       int min = A[0]; int p = 0;
3   ℓ0 : for (int i = 0; i < n; i++) {
4         if (A[i] < min) {
5           min = A[i]; p = i;
6         }
7       }
8       bool found = false;
9   ℓ1 : for (int i = 0; i < n; i++) {
10        assume min <= A[i];
11        if (A[i] == min) {
12          B[i] = 1; assume B[i] == 1;
13          found = true;
14        } else {
15          B[i] = 0; assume B[i] == 0;
16        }
17      }
18      assume found;
19      return min;
20  }
```

Figure 6: Pseudocode for T_I of Find Min. The code in red is the augmentation needed for proving $T_I \leq T_E$.

tion in constraint size, and explain the key insights behind the refinement proofs.

5.1 Find Min

Given a non-empty array A of length n, the problem is to find its smallest element, min, and mark all occurrences of the minimum using another array B. More precisely, there must exist an index p such that the following conditions hold:

1. $0 \leq p < n$ and $\texttt{min} = \texttt{A}[p]$,
2. for each $i \in [0,n)$, $\texttt{min} \leq \texttt{A}[i]$,
3. for each $i \in [0,n]$, $\texttt{B}[i] = (\texttt{A}[i] = \texttt{min}\,?\,1 : 0)$.

T_S encodes this specification by nondeterministically choosing min, each $\texttt{B}[i]$, and p. It uses two loops that iterate over A to enforce conditions 2 and 3.

T_I is shown in Fig. 6 (without the code in red, which we will discuss later). It also requires two loops: ℓ_0 to compute min, and ℓ_1 to compute the $\texttt{B}[i]$. Comparing T_I and T_S, we note that the two loops in T_I and T_S have exactly the same costs. However, T_S performs an additional dynamic LOAD, namely $\texttt{A}[p]$, to enforce Condition 1. Hence, $\mathcal{C}_{T_S}$ incurs the extra cost of RAM initialization, which performs n STOREs to write A into the memory, and is therefore larger than $\mathcal{C}_{T_I}$.

However, we can do better than either T_S or T_I. First, observe that unlike in T_I, we can merge the two loops in T_S for conditions 2 and 3 into a single loop because min can be chosen nondeterministically upfront. Compared to T_I, this saves one of the two inequality tests $i < n$ that $\mathcal{C}_{T_I}$ would otherwise include for each iteration of the two loops. Furthermore, we can eliminate the LOAD $\texttt{A}[p]$ in Condition 1 of T_S by introducing an auxiliary variable found that indicates whether min has been encountered at least once in the loop that checks conditions 2 and 3. The pseudocode of the resulting T_E is shown in Fig. 7 (excluding the blue code, which we will use later to establish that T_E refines T_S).

Thus, $\mathcal{C}_{T_E}$ needs only $2 \cdot n$ inequality tests, saving $1/3$ over

```
1   int find_min_efficient(n, A, B) {
2       int min, p;
3   ℓ'0 : havoc min;
4       bool found = false;
5   ℓ'1 : for (int i = 0; i < n; i++) {
6         assume min <= A[i];
7         if (min == A[i]) {
8           havoc B[i]; assume B[i] == 1;
9           found = true; p = i;
10        } else {
11          havoc B[i]; assume B[i] == 0;
12        }
13      }
14      assume found;
15      assume 0 <= p < n && A[p] == min;
16      return min;
17  }
```

Figure 7: Pseudocode for T_E of Find Min. The code in blue is the augmentation needed for proving $T_E \leq T_S$.

$\mathcal{C}_{T_I}$. Since the encoding of inequality tests dominates the size of the generated constraints, we observe a similar constant factor improvement in the overall constraint size.

Turning to the refinement proofs, if we add the red code in Fig. 6 to T_I, we obtain the augmented transition system $\hat{T}_{IE}$ for showing $T_I \leq T_E$ (see §4.2). Recall that the main part of the refinement proof is to show that the added **assume** statements in $\hat{T}_{IE}$ coming from T_E always succeed. We focus on the **assume** on Line 18, which is the most interesting one. Observe that the loop at ℓ_0 ensures $0 \leq p < n \wedge \texttt{A}[p] = \texttt{min}$ after the loop has terminated. Using this fact, we can then establish the loop invariant $i < p \vee$ found for the second loop at ℓ_1. This then allows us to prove that the **assume** statement on Line 18 is safe.

Next consider the refinement $T_E \leq T_S$. Adding the blue code in Fig. 7 to T_E yields an augmented transition system $\hat{T}_{ES}$ for the refinement proof $T_E \leq T_S$. We focus on showing that T_E ensures Condition 1. (The other two conditions follow immediately from the loop in T_E.) To this end, we can establish the loop invariant $\texttt{found} = 0 \vee (0 \leq p < n \wedge \texttt{A}[p] = \texttt{min})$ for the loop at ℓ'_1. Together with Line 14, this implies that adding the assume on Line 15 is safe. This line then establishes Condition 1.

We note that we would not be able to improve over T_I if the array A was guaranteed to have a single minimum, or if we were satisfied with finding any of the minimums in A. The loops at ℓ_1 and ℓ'_1 would be unnecessary.

5.2 Binary Search

Given a sorted array A, the bounds l, r of a possibly empty segment in A, and a value x, the problem is to compute i such that $l \leq i \leq r$ and $A[i] = x$. If no such i exists, return $i = -1$.

T_S for this problem checks i according to the specification above. That is, if $i \neq -1$, T_S checks that $l \leq i < r$ and $x = A[i]$, otherwise it iterates over $A[l \ldots r]$ and checks that the segment does not contain x. T_I is based on standard binary search.

For our cost analysis we focus on the number of LOAD operations, which is the largest contributor to the size of the generated constraints. In the worst case, T_I performs $\log(n)$ LOAD opera-

tions to search through the segment $A[l \dots r]$ where $n = r - l$. In contrast, T_S performs $n+1$ LOAD operations in the worst case. That is, T_S is asymptotically worse than T_I.

We can do better by exploiting that A is sorted. Introducing an auxiliary value s, we divide the specification for the case when x is not present ($i = -1$) into four subcases while retaining the specification for the case when $i \neq -1$. The refined specification becomes:

1. If $i = -1$, then $l = r$ or $x < A[l]$ or $x > A[r-1]$ or $(l \leq s < r-1 \wedge A[s] < x < A[s+1])$,
2. else $l \leq i < r \wedge A[i] = x$.

T_E is the direct encoding of this case analysis. It performs a constant number of LOAD operations, achieving an asymptotic improvement over T_I. We note that if the search is viewed as a standalone program, then this improvement is overshadowed by the cost of storing the array segment into RAM, which is linear in n. However, if the search is executed many times or viewed as a subroutine, then the RAM initialization can be amortized.

For proving $T_E \leq T_S$, observe that each of the subcases of Condition 1 implies that x cannot be present anywhere in the segment. For the last three cases, the proof relies on the precondition that A is sorted in strictly increasing order. For proving $T_I \leq T_E$, recall that binary search iteratively shrinks a subsegment $A[l' \dots r']$ of $A[l \dots r]$ that may still contain x. This process continues until the subsegment converges to a single point $l' = r'$, which is the index of the least element larger than x. In the nontrivial case where $l \neq r$ and x is not present in the segment but within the range of values defined by $A[l]$ and $A[r-1]$, we define $s = l' - 1$ for the final point $l' = r'$. Then s is the index that satisfies the last disjunct in Condition 1.

5.3 2D Convex Hull

Given a set of points $P = \{p_0, \dots, p_{n-1}\} \subseteq \mathbb{Z}^2$ with $n > 1$, assume no three points are on the same line, the problem is to find all points in P that lie on the convex hull of the set.

We additionally require P to satisfy the precondition of Graham Search [46], a popular algorithm that solves the 2D Convex Hull problem. Specifically, p_0 has the smallest y coordinate among all points in P, and the greatest x coordinate among all points in p with the same y coordinate as p_0. The remaining points are sorted in counterclockwise order when using p_0 as a reference point. In other words, for each $i \in (0,n)$, let L_i be the line passing through p_0 and p_i. Then intersect L_i with a horizontal line at p_0 and define $\angle_i$ to be the *top-right* angle of the intersection. P is ordered so that $\angle_i < \angle_{i+1}$ for all i.

With these assumptions, C defines the convex hull of P iff the following conditions hold:

1. $C \subseteq P$.
2. $p_0 \in C$ and for all $i \in (0,n)$, $p_i \in C$ or the angle defined by the points $(\mathtt{prv}_i, p_i, \mathtt{nxt}_i)$ bends inwards, where $\mathtt{prv}_i$ and $\mathtt{nxt}_i$ are the first points before and after p_i in P that are also in C. If no such $\mathtt{nxt}_i$ exists, then $\mathtt{nxt}_i = p_0$.

Condition 1 ensures that C contains no points outside P. Since P is sorted, Condition 2 guarantees that C contains all the points of P that lie on the convex hull of P.

```
1  int X_PROD(p, q, r) {
2    return (q.x-p.x) * (r.y-p.y) - (q.y-p.y) * (r.x-p.x)
3  }
4  void 2d_convex_hull_efficient(n, P, C) {
5    int k;
6    havoc k;
7    point nxt, prv;
8    havoc nxt; // nxt_0
9    prv = P[0]; // prv_1
10   havoc C[0]; assume C[0] == prv;
11   int count = 1;
12   for (int i = 1; i < n; i++) {
13     point cur = P[i];
14     if (nxt == cur) { // P[i] in C
15       havoc C[count]; assume C[count] == cur;
16       // get nxt_i because nxt_{i-1} ≠ nxt_i
17       havoc nxt; // nxt_i
18       // angle (prv, cur, nxt) must bend inwards
19       assume X_PROD(prv, cur, nxt) > 0;
20       prv = cur; // prv_{i+1}
21       count++;
22     } else { // P[i] !in C
23       // nxt_i = nxt_{i-1}; prv_{i+1} = prv_i
24       assume X_PROD(prv, cur, nxt) < 0;
25     }
26   }
27   assume nxt == P[0];
28   assume k == count;
29 }
```

Figure 8: Pseudocode of T_E for 2D Convex Hull.

T_S nondeterministically chooses C and then checks the above conditions. The size of $\mathcal{C}_{T_S}$ is in $O(n^2)$. (In particular, for each p_i, T_S needs to iterate through P again to find $\mathtt{prv}_i$ and $\mathtt{nxt}_i$.).

We use Graham Search as the T_I for this problem. For each of the n points, T_I needs two STORE operations and two dynamic LOAD operations.

T_E is shown in Fig. 8. It nondeterministically chooses k, C, and the points $\mathtt{nxt}_i$, then it iterates over the p_i and checks all relevant conditions in constant time for each i. The refinement proof showing $T_I \leq T_S$ uses the fact that T_I computes the points in C in the order in which they appear in P. Moreover, the $\mathtt{nxt}_i$ can be computed by T_{IE} using a simple linear scan of the final C.

To see that T_E yields smaller constraints than T_I, observe that only the array access of C[count] on Line 15 is dynamic and incurs the cost of two LOADs (one for each coordinate of the point). Also, there are no STORE operations. (Recall that a **havoc** command stands for augmented code in T_{IE}. Hence, it does not contribute to $\mathcal{C}_{T_E}$.) So T_E only performs two dynamic LOAD operations per iteration. The cost of a STORE depends on how deeply it is nested in conditionals whereas the cost of a LOAD does not [92, §3.1]. Specifically, each STORE in T_I is four times more expensive than a LOAD in T_E. We therefore expect that the size of $\mathcal{C}_{T_E}$ is about five times smaller than that of $\mathcal{C}_{T_I}$.

If we consider the variant where the problem is not to enumerate C but to compute its characteristic function on the indices of P (that is, mark the points in P that belong to C), then we can eliminate all dynamic LOAD operations from T_E and achieve an

asymptotic $\log(n)$ factor improvement over T_I.

5.4 Maximal Strong Components

Given a directed graph $G = (V, E)$ with nodes V and edges $E \subseteq V \times V$, the problem is to partition V into the maximal strongly connected components $C_0, \ldots, C_{k-1}$ of G. We represent the C_i implicitly using an array `rank` that maps every node $v \in V$ to the index of its maximal strongly connected component. That is, we define for all $i \in [0,k)$, $C_i = \{v \in V \mid \texttt{rank}[v] = i\}$. Given this, the formal problem statement is to find k and `rank` such that the following three conditions hold:

1. For all $v \in V$, $0 \leq \texttt{rank}[v] < k$.
2. For all $i \in [0,k)$, there exists a cycle c_i in G that visits exactly the nodes in C_i.
3. For all $i \in [0,k)$ and all cycles c in G, if c visits some node in C_i then c visits only nodes in C_i.

T_S encodes the above specification by nondeterministically choosing k, $\texttt{rank}[v]$ for each node $v \in V$, and the cycles c_i for each component $i \in [0,k)$. Condition 3 quantifies over the set of all cycles in G, which is in general an infinite set. However, it can be shown that restricting the quantification to *simple* cycles in G yields an equivalent condition. A simple cycle is a path where only the first and last node are equal and all other nodes are distinct. The condition that quantifies over simple cycles can be encoded using a nested loop that iterates over all partial permutations p of nodes in G and then checks that if p forms a simple cycle in G and intersects with a C_i, then it is fully contained in C_i. As the number of partial permutations grows exponentially with $|V|$, so does $|C_{T_S}|$.

We use Dijkstra's MSC algorithm [33, Chapter 25] as our T_I. The algorithm iterates over E and V. In each iteration, it performs up to 13 `LOAD` and 8 `STORE` operations. These operations dominate the size of the generated constraints.

However, we can again construct a T_E that improves over both T_I and T_S. The key idea for T_E comes from Dijkstra's correctness argument for his algorithm. Dijkstra observed that a set of connected components $C_0, \ldots, C_{k-1}$ is maximal iff it can be ordered so that all edges leaving a C_i target only nodes in components preceding C_i. Given Dijkstra's observation, we can replace Condition 3 in T_S with the following condition in T_E:

3*. For all $(v,w) \in E$, $\texttt{rank}[w] \leq \texttt{rank}[v]$.

Replacing Condition 3 by 3* yields a refinement of T_S.

Additionally, Condition 2 can be reformulated as a Condition 2* that no longer relies on the construction of explicit cycles c_i connecting the nodes in each component. We observe that the nodes in each C_i can be arranged in a tree that implicitly witnesses the existence of an appropriate c_i (which we use in the $T_E \leq T_S$ proof). The tree reflects the way T_I traverses the nodes in V and collapses candidate components whenever a node is revisited. These trees can be obtained from T_I using an augmentation that does not increase T_I's asymptotic complexity. We use this augmentation to establish Condition 2* when proving $T_I \leq T_E$.

Further details, including how Condition 2* is expressed, are described in Appendix B. What is important is that the combined size of these trees is linear in $|V|$ and so is checking their correctness. As a result, for dense graphs ($|E| \approx |V|^2$), the cost to enforce Condition 2* is insignificant. A detailed cost analysis yields an expected reduction in total constraint size for dense graphs by a factor of 17.5 for sufficiently large $|E|$. For shallow graphs ($|E| \approx |V|$), we still obtain a reduction by a factor of two. The principal savings come from the fact that conditions 3* and 2* can be checked by T_E (in the sense of validated inside an **`assume`**) with many fewer `LOAD` and `STORE` operations versus T_I.

Dijkstra's MSC algorithm is similar to Tarjan's algorithm [86]. We note that earlier work [24] already proposed an efficient checker for certifying the output of Tarjan's algorithm. Their approach shares with ours that it constructs trees from the graph to efficiently check whether the computed components are connected. However, the details of how these trees guarantee the existence of a cycle for each component differ from the trees used by our T_E. Moreover, their approach does not immediately yield an efficient encoding into constraints.

5.5 Minimum Spanning Tree

Given a connected graph $G = (V, E)$ where undirected edges have unique positive weights, the problem is to find M, the unique minimum weight connected spanning subgraph of G. M is called the minimum spanning tree (MST) of G. A natural, yet crude, specification is: M is a set of $|V| - 1$ edges that is connected and spanning, and all other sets of $|V| - 1$ edges are either not connected, not spanning, or heavier than M. The T_S that would encode this specification is exponential in $|V|$ because it needs to consider all $\binom{|E|}{|V|-1}$ candidates for M.

We will use an alternate definition for MST that leads to a more efficient T_S. Specifically, an MST, M, is the unique set of all edges that are not the heaviest in any cycle [77]. There are exactly $|V| - 1$ edges with this property. Thus, for all edges $e \in E \setminus M$, e is the heaviest in at least one cycle. Our T_S encodes this specification by nondeterministically picking an alleged MST $\tilde{M}$ and then for each $e \in E \setminus \tilde{M}$ (there are $|E| - |V| + 1$ such edges), providing a cycle where e is heaviest. Notice that there is no need to explicitly consider edges $e \in \tilde{M}$: after eliminating all $|E| - |V| + 1$ edges that are heaviest in some cycle, the remaining $|V| - 1$ edges ($\tilde{M}$) are the unique MST. Cycles are $O(|V|)$ edges in the worst case, and there are $O(|E|)$ edges outside of M, so the complexity of this T_S is dominated by $O(|V| \cdot |E|)$ edge lookups.

We use Kruskal's algorithm [53] as our T_I. It starts with M empty, sorts edges by weight, and iteratively adds edges to M if they don't form a cycle. This algorithm uses a Disjoint Set data structure to keep track of components of M and detect cycles. This data structure forms a partition of V into equivalence classes where two vertices are in the same class if they are connected by edges that have already been considered by the algorithm. Thus, when considering whether an edge e does or doesn't form a cycle with previous edges, Kruskal's algorithm need only check whether both endpoints of e are in the same equivalence class; if so, e is not added to M, and if not, e is added and the equivalence classes are merged.

The specific operations supported by a Disjoint Set data structure are:

- `MAKE-SET(v)`: turn vertex `v` into a singleton set.

- FIND-SET(v): return a unique identifier (root vertex) for the set containing v; also, re-parent all vertices on the path from v to the root to point directly to the root.
- UNION(u, v): take two different set identifiers (vertices u and v) and join the two sets together. Some bookkeeping happens to minimize the depth of the union, which keeps FIND-SET calls cheap.

MAKE-SET and UNION both use $O(1)$ memory operations. We use an implementation [87] of this data structure in which FIND-SET(v) has an average-case complexity of $O(\alpha(|V|))$ memory operations (where α is the inverse Ackermann function) and a worst-case complexity of $O(\log|V|)$ memory operations.

Given the amortized complexity of FIND-SET(v), there are two ways to compile T_I to constraints. One option is to unroll all FIND-SET(v) operations to their worst case $O(\log|V|)$ bounds. An alternative is to collect all nested loops into a state machine (as described in Buffet [92, §4]). The former results in $O(|E|\cdot\log|V|)$ RAM operations (§2) with a small constant. The latter has better asymptotics, only requiring $O(|E|\cdot\alpha(|V|))$ iterations of the state machine. However, the constant is large, because the state machine has many states (at least five, as detailed in Appendix C), each requiring multiple RAM operations.

Our T_E achieves both good asymptotics *and* a small constant. It builds on the idea behind widgets—checking FIND-SET rather than actually executing its logic—and introduces other techniques. The techniques are more fully described in Appendix C. At a high level, our T_E nondeterministically receives the MST, M, and the history, H, of the Disjoint Set operations; its constraints check the validity of M and H with respect to the input, the algorithm, and the data structure specification. This approach can be understood as directly encoding a special-purpose memory, namely the Disjoint Set data structure, as opposed to implementing that data structure on a general-purpose RAM (§2).

T_E represents H as a table of tuples, where each tuple contains: the operation being performed (MAKE-SET, FIND-SET, UNION, and UPDATE, which is a new operation that abstracts steps of the implementation of FIND-SET, described further below), a vertex, its old parent, its new parent, and the weight of the edge being examined by this operation.

To check H and M, T_E needs to:

1. Check that M is a $(|V|-1)$-sized subset of E;
2. Check H is consistent by verifying consistency between the old and new parent in consecutive operations on the same vertex;
3. Check that the data structure is consistent by ensuring MAKE-SET, FIND-SET, and UNION behave correctly and preserve the invariants of the Disjoint Set;
4. Check that for each edge e not in M, H reports that the endpoints of e are in the same set; and
5. Check that for all edges in M, H reports that the endpoints of each edge are in different sets and that the history merges those sets.

As an example, we elaborate on how to check that all FIND-SET operations are consistent (one component of the third check). Recall from earlier that a FIND-SET involves a sequence of re-parent operations. T_E encodes that sequence using the aforementioned UPDATE. We now define UPDATE by way of an example. Consider a tuple with $(\texttt{UPDATE}, \texttt{u}, \texttt{v}, \texttt{w}, 23)$. The meaning of this tuple appearing in H is that the prover is claiming that at the moment that the edge e with weight 23 was considered, one of e's endpoints was u, which had parent v in the Disjoint Set data structure; the parent of u was then immediately rewritten to be w.

Now, for a given FIND-SET operation to be validated, one requires the following. First, there is a sequence of tuples that starts with the corresponding FIND-SET tuple in H, immediately followed by zero or more consecutive UPDATE tuples. Second, for all tuples in the sequence (including the initial FIND-SET), the old parent must be the vertex of the next tuple in the sequence (informally: the algorithm is progressing toward the root). Third, in those tuples, the new parent must be the vertex of the last tuple in the sequence (informally: the algorithm re-parents consistently). Fourth, the last tuple in the sequence must have the vertex, old parent, and new parent all equal to each other (informally: the sequence ends at a root). To be clear, these properties are necessary but not sufficient to validate FIND-SET; another requirement is that the other numbered steps above hold (not just step 3), for example, H must be consistent. Of course, constraints that encode T_E enforce all of these properties and conditions.

We turn now to T_E's qualitative costs. At a high level, T_E walks down a table of tuples. The asymptotically high-order cost from doing so is handling a collection of $\Theta(|E|\cdot\alpha(|V|))$ UPDATE and FIND-SET operations. The constraint translation of this logic has a much lower constant than in the state machine approach because the conditionality has only two branches (UPDATE vs. FIND-SET; see Appendix C). Just as important, neither branch has RAM operations; instead, this table encodes, in and of itself, the history of operations on a special-purpose memory. That is, T_E semantically understands "update on the Disjoint Set", so T_E avoids explicit program logic (conditional statements, LOADs, STOREs) that implements Disjoint Set operations.

Quantitatively, we make this point by comparing the core approach in T_E—encoding the Disjoint Set as its own primitive—to encoding the history of Disjoint Set on top of RAM (§2), specifically Buffet-style RAM [92, §3]. Individual Disjoint Set operations in T_E are $2.5\times$ more expensive than RAM operations because they use 5-tuples instead of 4-tuples (a $1.25\times$ increase) and require 2 full transcript sorts instead of just 1 (a $2\times$ increase). However, these increases are swamped by savings from removing the need for nested loop unrolling or a state machine. On all input sizes, T_E outperforms both versions of T_I. On large inputs, T_I with a state machine outperforms T_I with loop unrolling; on such inputs, T_E requires $52.2\times$ fewer constraints than the better T_I. T_E also sees additive improvements when $|E|\gg|V|$; these are due to further techniques described in Appendix C.

Although we have focused on the specific example of the Disjoint Set data structure, as used by Kruskal's algorithm, the technique introduced here is much more general: it applies to any amortized data structure.

6 Experimental evaluation

This section answers the following questions:

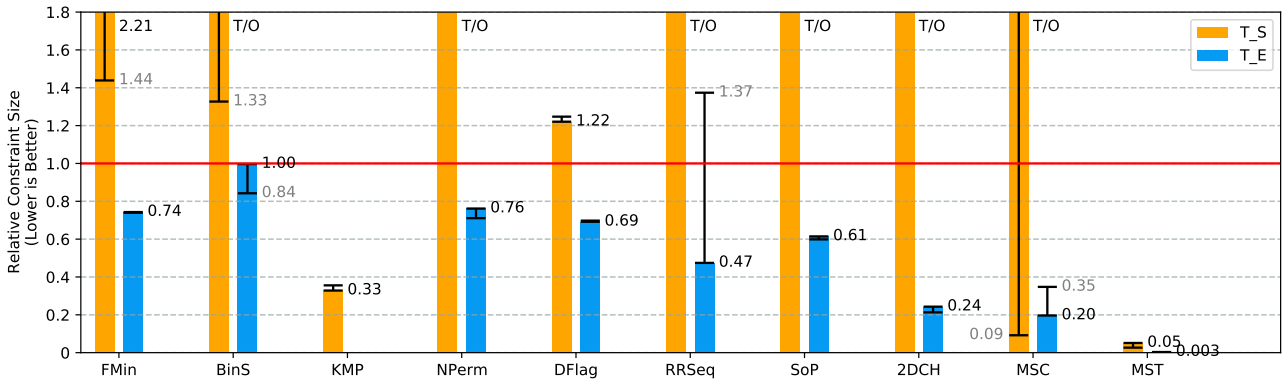
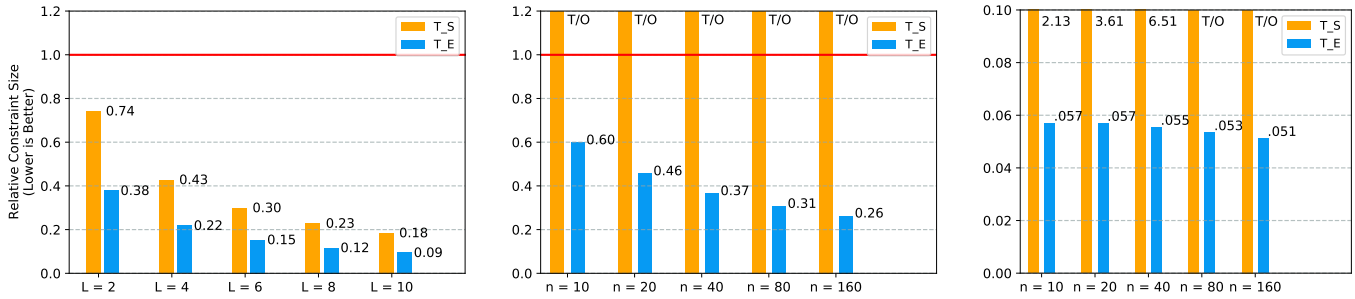


Figure 9: Relative $|\mathcal{C}|$ for T_S (orange) and T_E (blue) compared to the baseline T_I (red). The graph shows the problems where $\mathcal{C}_{T_E}$ improves over $\mathcal{C}_{T_I}$ by a constant factor (in the limit). The columns show the measurements obtained for the largest problem instances for which Pequin is able to compile T_I without timing out. In many benchmarks, the run time of T_S of the largest problem instance exceeds the timeout threshold. We use "T/O" to denote these cases. The error bars indicate the spread of the measurements obtained for smaller problem instances. For the Pattern Matching problem (KMP) we have $T_S = T_E$.



(a) Merge

(b) Binary Search (w/o RAM init.)

(c) 2D Convex Hull (marking nodes)

Figure 10: Relative $|\mathcal{C}|$ for T_S (orange) and T_E (blue) compared to the baseline T_I (red) for the problems where $\mathcal{C}_{T_E}$ improves asymptotically over $\mathcal{C}_{T_I}$ with increasing input size. T_I is omitted for the variant of the 2D Convex Hull problem considered here because the improvement is so vast.

(1) How difficult is it to build an end-to-end system for probabilistic proof checking based on Distiller?
(2) Does Distiller increase confidence in the correctness of widgets?
(3) Can we empirically achieve a constraint size reduction when using Distiller with existing front-ends?

End-to-end system. To answer the first question, we implement our framework (§4) in a system also called Distiller. The system takes T_S, T_E, and T_I as input. The system partially automates the refinement proofs and implements the new probabilistic proof pipeline proposed in Section 4.2. The input transition systems are expressed in a simple imperative programming language. Distiller's input format enables the user to augment these transition systems with ghost code to be used in the refinement proofs, for Correctness and for Coupling.

To check Correctness, the system generates skeletons of the refinement proofs $T_I \leq T_E$ and $T_E \leq T_S$ from its input. The proof skeletons are expressed in the Viper intermediate verification language [68]. The user can augment these proof skeletons with proof annotations (e.g. loop invariants) and then check them using the Viper verification tool. For the proof of $T_I \leq T_E$, the system computes the transition system $\hat{T}_{IE}$ by replacing all **`assume`** statements coming from T_E by **`assert`** statements. Viper checks that these **`assert`** statements are safe. The tool also checks that the proof annotations are correct. In particular, it checks that all user-provided loop invariants are indeed inductive. Distiller proceeds similarly for the proof of $T_E \leq T_S$.

To enable Coupling, Distiller takes T_E from its input and translates it to the language accepted by the Pequin toolchain [76], which is a subset of C, extended with domain-specific constructs. Distiller also takes $\hat{T}_{IE}$ from the previous step and extracts the program T_{IE}, which it translates to a standard C program. The two generated programs are then fed into Pequin. Pequin in turn compiles T_E to constraints $\mathcal{C}_{T_E}$, runs T_{IE}, and feeds the values into $\mathcal{C}_{T_E}$.

One of the constructs that Pequin supports is assertions. Each assertion translates to R1CS constraints checking that the assertion holds. Distiller uses this construct to compile the **`assume`** statements in T_E. Another Pequin construct is `exo_compute`, a hook allowing the prover to execute a program that produces values for arbitrary nondeterministic inputs to the generated con-

straints. This feature enables the prover to run T_{IE} and supply the auxiliary inputs to $\mathcal{C}_{T_E}$ when solving the constraints.

We perform an end-to-end evaluation of the resulting probabilistic proof pipeline composed of Distiller and Pequin for a select subset of our benchmarks. As a basic test of End-to-end Completeness, we apply the pipeline to the encoded benchmarks and successfully run it on a range of inputs. We note that the overhead of executing T_{IE} versus T_I in the solving step is negligible compared to the rest of the pipeline (recall that Step 3 contributes negligibly to costs in the first place; §2). As a basic test of End-to-end Soundness, we also run Distiller with buggy versions of the T_{IE}. In these cases, the back-end correctly rejects the computation.

Improved Reliability. To answer Question (2) we check the $T_I \leq T_E$ and $T_E \leq T_S$ proofs for 10 of our 11 benchmark problems (§5) using Viper. We omit the MST benchmark in this experiment because, here, T_S also encompasses the specification of a refined RAM, making the proof mechanization more elaborate.

Viper verifies all proofs. However, for two of the benchmarks we discovered bugs in the initial version of T_E. These bugs would have compromised End-to-end Soundness (§2). One bug was a missing array bounds check in T_E of the merge computation (§3). The other one was a subtle omission of a check in T_E for the Sum of Powers problem [33, Chapter 19]. We discovered these bugs when trying to annotate the respective T_E to prove $T_E \leq T_S$.

Constraint size reduction. Recall that our primary performance metric is $|\mathcal{C}|$ (§2). Our final experiment assesses Distiller against this metric. For all of our benchmarks, we generate T_I, T_E, and T_S as input programs for Pequin to be compiled to constraints. Then, with the exception of MST, we run Pequin's front-end on all three programs for a range of values for the loop unrolling bound that determines the maximal input size for each benchmark problem. As MST relies on a refined RAM construct that is not available in Pequin, we calculate the size of the constraints generated by all RAM operations by hand and run Pequin on the rest of the program. We then combine the result of the two parts to obtain the final constraint size. We enforce a timeout threshold of 240s per run, with the exception of MSC and MST, where a 2000s threshold is chosen to enable computations on larger problem instance sizes that demonstrate the exponential behavior of T_S. For each successful run, we measure the size of the generated R1CS constraints and compute the relative sizes of $\mathcal{C}_{T_S}$ and $\mathcal{C}_{T_E}$ compared to $\mathcal{C}_{T_I}$.

Figure 9 shows the results for the benchmarks where our theoretical analysis yields an improvement of T_E over T_I that converges to a constant factor with increasing problem instance size (Fig. 5). The results closely match our analysis. We note that for the MSC problem (§5.4), the relative improvement between T_E and T_I increases with the problem instance size. The maximal MSC instance size for which the translation of T_I does not time out is $n = 20, m = 400$. This is still too small to observe the $17.5\times$ theoretical improvement that we predict for dense graphs. Conversely, for the MST problem T_I has a large constant overhead that causes the improvement achieved with T_E to be $3\times$ larger on small instances than the predicted $52.5\times$ improvement for large problem instances. Finally, for binary search (BinS) we observe that the cost of storing the input array A into RAM, which is linear in the size of the array, dwarfs the $\log(n)$ improvement achieved for a single invocation of the binary search (§5.2).

Figure 10 shows the results obtained for the three problems where our theoretical analysis predicts that T_E performs asymptotically better than T_I with increasing problem instance size. The experiment again confirms our predictions. In particular, for the merge problem discussed in §3, Figure 10a shows that $|\mathcal{C}_{T_E}|/|\mathcal{C}_{T_I}|$ is approximately hyperbolic, which we expect because the predicted improvement for each point is $L\times$, where L is the number of merged arrays. Also, if we discount the RAM initialization cost for binary search, then we see the expected $\log(n)$ factor improvement (Fig. 10b).

7 Other related work

Probabilistic proofs. Section 2 gave an overview of probabilistic proof implementations, covering back-ends and front-ends; see also [88, 91, 94]. Unlike Distiller, none of the front-ends achieves all three requirements stated in the Introduction; in fact, none creates a framework for proving the correctness of widgets.

Distiller combines formal methods and probabilistic proofs. Very few works live at this intersection. Some notable exceptions are as follows. CirC [70] is a toolkit for building compilers to a family of constraint formalisms, including R1CS and SMT instances. Its architecture takes advantage of the rich SMT toolbox, allowing users to build powerful analyses and optimizations. Additionally, CirC includes partially verified compiler passes to help ensure soundness and completeness are preserved when optimizing high-level program statements into low-level constraints [71]. The two works are complementary: one could compile a Distiller-verified widget in CirC, to get further reductions.

The Orbis Specification Language (OSL) [89] has a similar ideology to ours: replace a computation with its formal specification, and compile the latter to constraints, in the hope of gaining more concise constraints. However, as our examples (§5) make clear, the cost of a naive specification is often exorbitant. So one has to identify an "in-between" specification, and (a) relate it to the abstract specification, and (b) derive an implementation that knows how to satisfy the in-between specification or the original. Neither of these problems is addressed by OSL. The authors mention that they want to synthesize implementations from specifications. Though, for the rich specification language we consider (general transition systems), whether a specification even *has* an implementation is undecidable. Thus, to instantiate the ideology that OSL and we share, one needs human input (to write down T_E, and relate it to the specification and the implementation).

In an under-appreciated work, Fournet et al. [38] develop a compiler, based on CompCert [59], that formally connects the semantics of a higher-level program to the constraint formalism (specifically R1CS constraints). This work is complementary to Distiller—it provides Translation Fidelity (§2).

Leo [30] also has the goal of formally verified translations to constraints. Leo develops a compiler and uses the ACL2 [49] theorem prover to validate each stage of translation. However, this falls short of a verified compiler, as in Fournet et al. Moreover, the authors of Leo want to validate hand-crafted gadgets. It is not

clear how to do this, since ACL2 cannot easily "reverse" R1CS instances to lift them to higher-level semantics. As a consequence, crucial pieces of verification are works in progress [30, §6.4]. By contrast, Distiller incorporates widgets soundly and completely, by treating them at the source code level and using refinement.

QED^2 [72] is another work at the intersection of formal methods and probabilistic proofs. This work is concerned with detecting underconstrained circuits (constraint systems where an input can have multiple outputs). They build a tool that checks if a particular set of constraints is fully constrained (and produces two satisfying outputs for a single input if not). We see this work as orthogonal to Distiller because QED^2 does not verify the connection between a specification and constraints. In particular, it is possible for a constraint system to be satisfied by something which doesn't satisfy the specification or vice versa. Additionally, the requirement that circuits correspond to functions (unique output for each input) makes the overall framework less expressive than the transition systems, traces, and refinements used by Distiller.

Another orthogonal work that combines formal methods and probabilistic proofs is zero knowledge abstract interpretation [36]. Here, the problem is to devise a scheme that enables a prover to convince a verifier of the result of a static program analysis run without revealing the analyzed program.

Refinement. The idea of developing a program from a specification in a step-wise refinement process goes back to early work by Dijkstra [32, 33] and Wirth [98]. The formal concept of refinement relations and mappings to relate the observable behaviors of transition systems was first explored in the 1980s [55, 56, 62]. It is a cornerstone of modern Formal Methods; applications include reasoning about concurrent and distributed systems, establishing program equivalence, and verifying security properties.

Abadi and Lamport [1] showed that refinement mappings yield a complete proof technique for establishing refinement. Though, in general, the technique requires the transition systems to be augmented with *history* variables (recording information about past states) and *prophecy* variables (predicting information about future states). Other related proof techniques for establishing refinement, for instance, based on (weak) simulation relations [65, 73, 90] are less suited for our purposes as they do not immediately provide a blueprint for computing satisfying assignments.

The notion of refinement considered in Distiller takes a monolithic view of transition systems, which makes it difficult to reason compositionally about subroutines. *Contextual refinement* [37] relates the observable behavior of subroutines subject to all possible client programs that may use them, thereby enabling compositional reasoning. For the relatively simple programming models supported by most existing probabilistic proof front-ends (no concurrency, object-oriented features, or higher-order functions), considering contextual refinement instead of *global refinement* does not add substantial complexity to the verification effort.

Distiller uses *mechanized proofs* (§6). Specifically, it uses a lightweight encoding of refinement proofs in the language of the deductive program verifier Viper [68]. The proofs are partially automated using SMT solvers. However, this is a choice. Nothing in our approach precludes or necessitates particular approaches to mechanization. In particular, there is a large body of work on refinement calculi that mechanize the correct step-wise refinement of programs and system models [2, 10, 64, 66, 67]. More recently, the many applications of proofs concerning products and couplings of two or more programs have fueled the development of relational program logics that provide frameworks for proof mechanization [11–13, 21, 39, 85, 100]. Several of these formal reasoning systems have been implemented in tools, including for instance TLA+ [57], Rodin [3], EasyCrypt [14], and ReLoC [39].

8 Discussion and conclusion

Distiller improves a key metric in implementations of probabilistic proofs, namely the number of arithmetic constraints required to encode the validity of the execution of a computation. The improvements typically range from small integer factors to orders of magnitude, depending on the computation. Distiller also introduces, for the first time, a framework for widgets that are correct by construction. This framework radically expands the space of potential widgets, thereby allowing probabilistic proofs to do much more, by paying much less.

The primary remaining verification gap is that we do not verify our tools, including the translator to the two targets, Pequin and Viper itself. This is not a fundamental limitation. In fact, we are encouraged by certified compiler work in this research area [38] to guarantee Translation Fidelity (§2). The TCB can be further reduced by using a verification toolchain with a small trusted core [6] (at the expense of reduced proof automation), or by deploying validation techniques that produce certificates for automatically generated proofs [34, 75].

Another limitation is that widgets are constructed manually. Identifying a T_E that slashes constraint size relative to T_S and T_E, and then proving its correctness can take significant time and effort. A promising direction for future work is to adapt techniques from program synthesis [4] and superoptimizing compilers [48, 84] to automate these steps.

The code for Distiller is available at: `https://github.com/PepperSieve/vprexocompiler`

Acknowledgments

We thank Sebastian Angel, Jacob Salzberg, Justin Thaler, and Riad Wahby for helpful conversations. This research was supported by DARPA under Agreement No. HR00112020022, NSF under grant CNS-1514422, and AFOSR under grant FA9550-18-1-0421. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government, DARPA, NSF, or AFOSR.

A Details of refinement mapping: merging

This section provides details on showing that the **`assume`** statements of $\hat{T}_{IE}$ (Fig. 4) of the merge computation express invariants of T_{IE}. This guarantees that the **`assume`** statements will always evaluate to true in T_I (Fig. 1), and completes the refinement $T_{IE} \leq \hat{T}_{IE}$. See Section 4.2 for the full discussion.

We justify the addition of the **assume** statements by providing supporting invariants at relevant portions of T_{IE}. Figure 11 shows $\hat{T}_{IE}$ with the desired invariants explicitly added. There are four **assume** statements added to T_{IE} that require justification, on Lines 33, 35, 37, and 38. We omit the proof of the **assume** statement on Line 11 as it is easy to show.

The first **assume** (Line 33) states that the most recently added element `B[i]` is properly ordered with respect to its preceding element. To prove this, we establish the invariants on Lines 21 to 23 at the inner loop, and the invariant on Line 15 at the outer loop. These invariants imply that, at each iteration of the outer loop, `B[i]` is set to a value greater than `B[i-1]`, provided `found` is `true` after the inner loop terminates. To show the latter, first observe that $\mathtt{k} = \mathtt{L}$ holds after the inner loop terminates. Now, if `found` were `false` at this point, then Line 20 would imply

$$\sum_{n=0}^{L-1} \mathtt{curr}[n] \geq \sum_{n=0}^{L-1} \mathtt{A_n.len}\ .$$

However, the invariants on Line 14 together with the loop condition `i < B.len` of the outer loop and the equation on Line 11 imply

$$\sum_{n=0}^{L-1} \mathtt{curr}[n] < \mathtt{B.len} = \sum_{n=0}^{L-1} \mathtt{A_n.len}\ .$$

This yields a contradiction. Hence, `found` must be `true` after the inner loop terminates.

To prove the second **assume** (Line 35), we can again use the invariant at Line 21 and the fact that `found` must hold after termination of the inner loop.

Similarly, the third `assume` (Line 37) follows from the invariants at Lines 21 and 22 at the inner loop, and 13 at the outer loop.

The final **assume** (Line 38) follows from the identity $\mathtt{running\_min} == \mathtt{A_{kstar}[curr[kstar]]}$. This is sufficient because `k_i` is set to `kstar`, `j_i` is set to `curr[kstar]`, and `B[i]` is set to `running_min`.

The remaining invariants that are not used directly to prove the **assume** statements are needed to ensure that the invariants are inductive. In particular, Line 24 at the inner loop is needed to show that Line 15 is maintained by each iteration of the outer loop.

The invariants, thus, justify the addition of the **assume** statements in $\hat{T}_{IE}$, proving the refinement $T_{IE} \leq \hat{T}_{IE}$.

B Details of MSC example

In this section, we expand on the discussion of the maximum strongly connected component problem in Section 5.4.

Given a directed graph $G = (V, E)$ with nodes V and edges $E \subseteq V \times V$, the problem is to partition V into the maximal strongly connected components $C_0, \ldots, C_{k-1}$ of G. We represent the C_i implicitly using an array `rank` that maps every node $v \in V$ to the index of its maximal strongly connected component. That is, we define for all $i \in [0, k)$

$$C_i = \{v \in V \mid \mathtt{rank}[v] = i\}\ .$$

Given this, the formal problem statement is to find k and `rank` such that the following three conditions hold:

1. For all $v \in V$, $0 \leq \mathtt{rank}[v] < k$.
2. For all $i \in [0, k)$, there exists a cycle c_i in G that visits exactly the nodes in C_i.
3. For all $i \in [0, k)$ and all cycles c in G, if c visits some node in C_i then c visits only nodes in C_i.

Weakest specification. T_S encodes the above specification by nondeterministically choosing k, $\mathtt{rank}[v]$ for each node $v \in V$, and the cycles c_i for each component $i \in [0, k)$. Condition 3, which enforces that the components C_i are maximal, is encoded by a nested loop, where the inner loop enumerates all cycles in G that visit exactly one node twice. As there are $O(|V|!)$ possible cycles that need to be considered, the size of the constraints generated from T_S grows exponentially in $|V|$.

Implementation We use Dijkstra's MSC algorithm [33, Chapter 25] as our T_I. The algorithm is similar to Tarjan's algorithm [86] but lends itself more directly to an efficient translation into constraints.

Dijkstra observed that a set of strongly connected components $C_0, \ldots, C_{k-1}$ is maximal iff it can be ordered so that all edges leaving a C_i target only nodes in components preceding C_i. That is, consider the graph $G_C = (V_C, E_C)$ such that $V_C = \{C_0, \ldots, C_{k-1}\}$ and $(C_i, C_j) \in E_C$ iff $i \neq j$ and $(v, w) \in E$ for some $v \in C_i$ and $w \in C_j$. Then the above ordering of the C_i is a (reverse) topological sort of G_C. The existence of this topological sort implies that G_C is acyclic and therefore the C_i cannot be joined into larger components. Given Dijkstra's observation, one can replace Condition 3 in T_S with the following:

3^*. For all $(v, w) \in E$, $\mathtt{rank}[w] \leq \mathtt{rank}[v]$.

The algorithm computes a `rank` that satisfies this revised condition. For pedagogy we discuss the abstract version of the algorithm shown in Figure 12, which assumes mathematical sets as an built-in type. Dijkstra's concrete version of the algorithm implements the relevant set operations using auxiliary variables to achieve a running time that is linear in the size of G.

The code highlighted in blue can be ignored for now. The algorithm iterates over V and E using two working sets `VW` and `EW` to keep track of the nodes and edges that still need to be processed. The stack `cc` is used to keep track of a chain of disjoint strongly connected components that are currently being traversed.

The algorithm maintains the invariant that each component is connected via an edge in G to its next component higher up on the stack. Moreover, the algorithm maintains that the set `VC` is the union of all the components in `cc`. A node leaves `VC` (and, hence, `cc`) once its maximal strongly connected component has been identified and its rank has been assigned. The variable `k` keeps track of the number of maximal components that have been identified so far.

Whenever the stack `cc` becomes empty, the outermost loop nondeterministically chooses an unprocessed node `v` from `VW` and adds it as a new singleton component to `cc` (Lines 14–17). While the stack is nonempty, the algorithm pops the topmost

```
1   void merge (L,A_0,..,A_{L-1},B) {
2   ℓ_0 : int[L] curr = {0};
3       int len, running_min, kstar, found, k_i, j_i;
4       len = 0;
5   ℓ_1 : for (int k = 0; k < L; k++)
6         invariant len == ∑_{n=0}^{k-1} A_n.len
7       {
8        len += A_k.len;
9       }
10      B.len = len;
11      assume B.len == ∑_{k=0}^{L-1} A_k.len;
12  ℓ_2 : for (int i = 0; i < B.len; i++)
13        invariant ∀ j :: 0 ≤ j < L ⇒ 0 ≤ curr[j]
14        invariant ∑_{n=0}^{L-1} curr[n] == i
15        invariant i == 0 ∨ ∀ j :: 0 ≤ j < L ∧ curr[j] < A_j.len ⇒ B[i-1] < A_j[curr[j]]
16      {
17        found = false;
18  ℓ_3 :   for (int k = 0; k < L; k++)
19          invariant !found ⇒ ∀ j :: 0 ≤ j < k ⇒ curr[j] ≥ A_j.len
20          invariant !found ⇒ ∑_{n=0}^{k-1} curr[n] ≥ ∑_{n=0}^{k-1} A_n.len
21          invariant found ⇒ 0 ≤ kstar < L
22          invariant found ⇒ curr[kstar] < A_kstar.len
23          invariant found ⇒ running_min == A_kstar[curr[kstar]]
24          invariant found ⇒ ∀ j :: 0 ≤ j < k ∧ j ≠ kstar ∧ curr[j] < A_j.len ⇒ running_min < A_j[curr[j]]
25        {
26          if (curr[k] < A_k.len && (!found || A_k[curr[k]] < running_min)) {
27            running_min = A_k[curr[k]];
28            kstar = k;
29            found = true;
30          }
31        }
32        B[i] = running_min;
33        assume i == 0 || B[i-1] < B[i];
34        k_i = kstar;
35        assume 0 <= k_i && k_i < L;
36        j_i = curr[kstar];
37        assume 0 <= j_i && j_i < A_{k_i}.len
38        assume B[i] == A_{k_i}[j_i];
39        curr[kstar]++;
40      }
41  ℓ_4: return;
42  }
```

Figure 11: Pseudocode for the transition system $\hat{T}_{IE}$ with invariants.

```
1  datatype comp =
2    Node(node)
3  | Rotate(node, comp, node)
4  | Concat(comp, comp)
5
6  int msc(V, E, rank) {
7    set<edge> EW = E;
8    set<node> VW = V;
9    set<node> VC = ∅
10   stack<set<node> * comp * node * node> cc = empty;
11   comp[] cycle;
12   int k = 0;
13   while (VW != ∅) {
14     int v = choose(VW);
15     VC = {v};
16     VW = VW \ {v};
17     push(cc, ⟨{v}, Node(v), v, v⟩);
18     do {
19       set<node> C, comp c, node i, _ = pop(cc);
20       while ({(u,w) ∈ EW | u ∈ C} != ∅) {
21         edge (u,w) = choose({(u,w) ∈ EW | u ∈ C});
22         EW = EW \ {(u,w)};
23         if (w ∈ VC \ C) {
24           // compact the chain
25           c = Rotate(i, c, u);
26           do {
27             set<node> C1, comp c1, node i1, node o1 =
28               pop(cc);
29             C = C ∪ C1;
30             i = w ∈ C1 ? w : i1;
31             c = Concat(Rotate(i, c1, o1), c);
32           } while (w ∉ C)
33         } else if (w ∈ VW) {
34           VW = VW \ {w};
35           VC = VC ∪ {w};
36           push(cc, ⟨C,i,c,u⟩);
37           C = {w}; c = Node(w); i = w;
38         }
39       }
40       for (w ∈ C) rank[w] = k;
41       cycle[k] = c;
42       k = k + 1;
43       VC = VC \ C;
44     } while (VC != ∅);
45   }
46   assert isMSC(V, E, k, rank, cycle);
47   return k;
48 }
```

Figure 12: Pseudocode for implementation T_I of MSC. The code in blue is the augmentation needed for proving $T_I \leq T_E$.

```
1  int msc_efficient(V, E, rank) {
2    comp[] cycle;
3    int k;
4    havoc k;
5    for (v ∈ V) havoc rank[v];
6    for (i ∈ [0,k)) havoc cycle[k];
7    assume isMSC(V, E, k, rank, cycle);
8    return k;
9  }
```

Figure 13: Pseudocode for T_E of MSC.

component C from cc. Then it attempts to nondeterministically choose an unprocessed edge (u,w) in EW such that its source node u is in C (Lines 21–22). If no such edge exists, then C must be maximal and its nodes are removed from VC (Lines 40–43). Otherwise, if the target node w of the chosen edge is in VC \ C, then by the invariant of cc (that every component in VC is pointing to the next component higher up on the stack), there exists a cycle connecting all the components in cc starting with the component that contains w, all the way to C. The loop on lines 26–32 thus compacts these components to a new component C. If on the other hand the target node w is an unprocessed node, then the current component C is added back to the stack and then C is updated to the new topmost component consisting of w (Lines 33–38). In all other cases, w is either already in C or is already in an identified maximal strongly connected component. In these cases, the edge can be discarded because it is guaranteed to satisfy Condition 3* based on the partial reverse topological sort that has already been computed.

To reduce the size of the generated constraints for T_I, we flatten the nested loops into a single loop that is executed $|V|+|E|$ times. In each iteration, T_I performs up to 13 LOAD and 8 STORE operations. These operations dominate the size of the generated constraints. In particular, the STORE operations are embedded under 4 layers of conditionals making them 16× more expensive than a LOAD.

Efficient specification. The idea behind T_E is to directly check the three conditions 1, 2, and 3*. Its pseudocode is shown in Figure 13. T_E nondeterministically assigns k, rank, and an auxiliary array cycle. The latter stores for each alleged connected component C_i, a cycle c_i that visits the nodes in C_i. The three conditions are assumed on Line 7 using the function isMSC. Before we discuss isMSC in detail, we argue $T_I \leq T_E$, which boils down to justifying the assume at Line 7 of Figure 13 by showing that the assert at Line 46 of Figure 12 always holds. The crux of the proof is to establish conditions 1, 2, and 3* as an invariant before the assert.

It is easy to see that T_I ensures Condition 1. To see that it ensures 3*, first note that when a component C is assigned its rank it is the topmost component in the chain cc. Moreover, its assigned rank k is larger than the rank of all previously assigned components. Hence, we must show that all edges leaving C point only to these *older* components. To this end, observe that the algorithm maintains the invariant that for the topmost component C in the chain, all edges starting in C that have already been processed either remain in C or target an older component. Since

C is assigned its rank only after all its edges have been processed, it follows that Condition 3* is satisfied.

For showing Condition 2, T_E nondeterministically chooses for each alleged connected component C_i, a cycle c_i that visits the nodes in C_i. Hence, to show $T_I \leq T_E$, we need to augment T_I with auxiliary code to compute the cycles stored in cycle. This is the code highlighted in blue in Figure 12.

The augmentation maintains for each connected component C in cc, an associated cycle c that visits all nodes in C. When C is assigned its rank, c is copied to cycle (Line 41). To maintain the invariant that c visits all nodes in C, the code must also construct a new cycle each time the algorithm computes a new component by compacting a cyclic chain in cc. Therefore, the augmentation additionally maintains for each component C, the target node i in C for the incoming edge from its predecessor in cc, and the source node o in C for the outgoing edge to its successor in cc (unless C is the topmost component, in which case o can be chosen arbitrarily).

Instead of representing each cycle c explicitly as a path in G, we represent it symbolically using an auxiliary tree-like datatype comp. To motivate this symbolic representation, consider the construction of an explicit cycle for a new compacted component obtained by the loop on lines 26–32. To this end, we need to construct a path c from w to u that visits all nodes in the new component. Since there exists an edge back from u to w, we obtain the desired cycle. The loop must therefore maintain the invariant that c is a path that visits all nodes in the components that have already been compacted into C. Moreover, this path must end in u and start in i. The latter is needed so that we can concatenate this path with another path that visits all nodes in the component C1 that precedes C in cc and that will be compacted next. The component C1 has an associated cycle c1 and is connected to its neighbors via i1 and o1. First, we *rotate* c1 to obtain a path that starts and ends in the node i1, thus visiting all nodes in C1 at least once. Next, we extend this path with the segment of c1 that goes from i1 to o1. Finally, we concatenate the resulting path with c using the edge $(\texttt{o1},\texttt{i})$ that must exist by the invariant of cc. The final path goes from i1 to u and visits all nodes in $\texttt{C1} \cup \texttt{C}$ (at which point i1 becomes the new i for $\texttt{C1} \cup \texttt{C}$), reestablishing the loop invariant.

The issue with this construction is that the size of the obtained cycle c can be exponential in the size of the compacted components. This blow-up would affect both the running time of the augmented T_I as well as the size of $\mathcal{C}_{T_E}$. By representing the cycles using the type comp, we avoid this blow-up. Intuitively, a comp value c is a program that provides instructions for constructing a cycle that visits all the nodes appearing in c. The size of this program is linear in the number of visited nodes. Moreover, checking whether a given comp program c indeed constructs a cycle that contains all the nodes appearing in c can be done in linear time. This checker is our missing ingredient for T_E to guarantee Condition 2.

The comp programs have a tree-like structure where each vertex p is labeled with an instruction to construct a path from the paths computed by p's children. There are three kinds of instructions, corresponding to the three different operations involved in the construction of cycles during compaction. The instruc-

```
1  bool isMSC(V, E, k, rank, cycle) {
2    bool b = true;
3    bool[V] seen = {false};
4
5    for (i ∈ [0,k))
6      b = b && isCycle(V, E, rank, cycle[i], seen, i);
7    for (v ∈ V) b = b && seen[v];
8    for ((u,v) ∈ E) b = b && rank[v] <= rank[u];
9
10   return b;
11 }
12
13 bool isCycle(V, E, i, rank, c, seen) {
14   match c {
15     case Node(n) =>
16       seen[n] = true;
17       return n ∈ V && rank[n] == i
18     case Rotate(l,c1,r) =>
19       return !seen[l] && !seen[r]
20         && isCycle(V, E, i, rank, c1, seen)
21         && seen[l] && seen[r]
22     case Concat(c1, c2) =>
23       return isPath(V, E, i, rank, c1, seen)
24         && isPath(V, E, i, rank, c2, seen)
25         && (right(c1),left(c2)) ∈ E
26         && (right(c2),left(c1)) ∈ E
27   }
28 }
29
30 bool isPath(V, E, i, rank, c, seen) {
31   match c {
32     case Node(n) =>
33       seen[n] = true;
34       return n ∈ V && rank[n] == i
35     case Rotate(l,c1,r) =>
36       return !seen[l] && !seen[r]
37         && isCycle(V, E, i, rank, c1, seen)
38         && seen[l] && seen[r]
39     case Concat(c1, c2) =>
40       return isPath(V, E, i, rank, c1, seen)
41         && isPath(V, E, i, rank, c2, seen)
42         && (right(c1),left(c2)) ∈ E
43   }
44 }
45
46 node left(c) {
47   match c {
48     case Node(n) => return n
49     case Rotate(l, _, _) => return l
50     case Concat(c1, _) => return left(c1)
51   }
52 }
53
54 node right(c) {
55   match c {
56     case Node(n) => return n
57     case Rotate(_, _, r) => return r
58     case Concat(_, c1) => return right(c1)
59   }
60 }
```

Figure 14: Pseudocode for checker used by T_E of MSC.

tion `Node(`n`)` for a node n constructs a (trivially cyclic) path that consists only of the node n. The instruction `Rotate(`l,c,r`)` constructs a new path from l to r that visits all nodes in c, provided c constructs a cycle that contains l and r. Finally, `Concat(`c_1,c_2`)` concatenates the two paths constructed by c_1 and c_2, provided there exists an edge in G from the end point of c_1 to the start point of c_2.

The implementation of `isMSC` is shown in Figure 14. It uses the function `isCycle` to check for each $i \in [0,k)$ whether `cycle[`i`]` constructs a cycle in G that visits all nodes appearing in `cycle[`i`]`. Moreover, it ensures that the `rank` of each of these nodes is indeed i. The auxiliary array `seen` is used to ensure that each node in V occurs in some `cycle[`i`]`.

We briefly discuss the $T_E \leq T_S$ refinement proof. The proof boils down to establishing conditions 1, 2, and 3 as an invariant after the `assume` at Line 7 of Figure 13. First, note that if `k` is negative, then the loop on Line 5 of Figure 14 is skipped. Hence, the next loop on Line 7 will set `b` to `false` because at least one node will not be seen, assuming V is nonempty. Therefore, the checker implies Condition 1. Next, to prove that `isCycle(`$V,E,$`rank,cycle[`i`],seen,`i`)` implies Condition 2 for component i, we construct an *interpreter* for `comp` programs. Given a `comp` program `c` that satisfies `isCycle`, the interpreter executes `c` to construct the cyclic path in G encoded by `c`. Finally, the loop on Line 8 directly checks Condition 3*. As we have argued earlier, this implies Condition 3.

The combined size of all `comp` programs is linear in $|V|$ and so is checking their correctness (Lines 5 and 6 in Figure 14). As a result, for dense graphs ($|E| \approx |V|^2$), the cost to enforce Condition 2 is insignificant. While checking Condition 3* is still linear in $|E|$, it involves only two `LOAD`s per edge and no `STORE`. A detailed cost analysis yields an expected reduction in total constraint size for dense graphs by a factor of 17.5 for sufficiently large $|E|$. For shallow graphs ($|E| \approx |V|$), we still obtain a reduction by a factor of two.

C Details of MST example

This section delves into the details of the Minimum Spanning Tree (MST) problem, to elaborate on Section 5.5. Pseudocode for T_I is shown in Figure 15; pseudocode for T_E is shown in Figures 16 and 17. Below, we explain how T_E is a widget for T_I.

We will do this by building and working directly with a *transcript of Disjoint Set operations* and by designing a checker that establishes that:

1. This transcript reflects the defined semantics of the Disjoint Set; and
2. This transcript corresponds to an execution of Kruskal's algorithm on a graph G.

Introducing H_{pre}. First, we instrument T_I to transcribe Disjoint Set function calls (`make_set`, `find_set`, and `union`; the interface is given in Section 5.5) with the information necessary to describe and simulate them. We represent entries in this transcript, which we call H_{pre}, as 5-tuples of the form:

(label, vertex, old parent, new parent, weight).

The *label* is one of `MAKE-SET`, `FIND-SET`, `UNION`, or `UPDATE`; it corresponds to the function being called. `MAKE-SET` and `UNION` correspond to calls to `make_set` and `union`, respectively. `FIND-SET` corresponds to a *top-level* call to `find_set` (Lines 40 and 41 in Figure 15). `UPDATE` corresponds to a *recursive* call to `find_set` (Line 16 in Figure 15).

The *vertex* field is an input to the function. For `make_set`, *vertex* refers to the vertex being created. For `find_set` (here and below, `find_set` refers to entries labeled `FIND-SET` and `UPDATE`), *vertex* refers to the vertex being queried. For `union`, *vertex* refers to the vertex with smaller rank.

The *old parent* field is redundant for `make_set`; for `find_set` and `union`, this is the parent of the vertex before the call modifies it. For all function calls, the *new parent* field is the parent of the vertex after the call modifies it.

The final field is *weight*, which is used for ordering Disjoint Set operations (§C.1) and associating them to edges in G (§C.2). For `make_set`, this field is -1 to ensure that they are ordered before all other operations. For `find_set` and `union`, this field is the weight of the edge being considered for inclusion in the MST when the call occurred.

This transcript only considers the Disjoint Set operations as acting on the `parent` array in T_I and ignores the `rank` array. This is because the `rank` array is useful only for preserving asymptotics (and not correctness). As a result, we can freely augment T_I, replacing accesses to `rank` with nondeterministic choice (Line 23 in Figure 15).

C.1 Checking Disjoint Set operations from H_{pre}

Given some H_{pre}, to check that it is a valid series of Disjoint Set operations, there are two key tests. First, a vertex's parent must be consistent from operation to operation (the new parent in the operation for a given vertex must appear as the vertex's old parent in the next operation), assuming the operations are executed in ascending order of weight. Second, the semantics of `make_set`, `find_set`, and `union` are preserved (and thus invariants on the structure of the Disjoint Set are preserved across operations). These two tests, namely parent consistency and Disjoint Set semantics, refer to Conditions 2 and 3 respectively in Section 5.5. We detail these tests next.

The parent consistency test is in `assume_memory_checks()` (Line 30 in Figure 17). For this test, the prover nondeterministically reorders H_{pre} as H_{par} and performs the following three checks for each consecutive pair of rows in H_{par}. First, the rows must be sorted into groups by vertex. Next, consecutive rows with the same vertex must be sorted by their weights. Finally, the old parent of row i must be the new parent of row $i-1$ if their vertices are the same. The first of the H_{par} checks is efficiently enforced by defining an order over the vertices and then checking that the rows are sorted by this order.

These checks are very similar to those for checking RAM [16, 18, 78, 92] in constraints. Here, (vertex, weight) is analogous to RAM's (address, timestamp), and parent is analogous to RAM's value. As a technical point, the timestamps in RAM create a total order on operations, whereas for this problem, weight imposes only a partial order: when a particular edge is being considered, there could be multiple operations on a given vertex, and the approach outlined here does not require that the operation order

```
1 struct DisjointSet {
2   int parent[MAX]; // indexes are not known statically
3   int rank[MAX];  // indexes are not known statically
4   DisjointSet(int num_v) {
5     for (int i = 0; i < num_v; i++) {
6       make_set(i);
7     }
8   }
9   void make_set(int i) {
10    parent[i] = i;
11    rank[i] = 0;
12  }
13  int find_set(int v) {
14    if (v != parent[v]) {
15      parent_v_tmp = parent[v]
16      new_root = find_set(parent_v_tmp);
17      parent[v] = new_root
18    }
19    return parent[v];
20  }
21  void union(int u, int v) {
22    // sort the two vertices by rank
23    bool swap = (rank[u] < rank[v]);
24    int min = swap * (u - v) + v;
25    int max = swap * (v - u) + u;
26    // join the two sets
27    parent[min] = max;
28    rank[max] = max(rank[max], rank[min] + 1);
29  }
30 }
31
32 Tree mst_kruskal(Graph G) {
33   DisjointSet ds = new DisjointSet(G.num_vertices);
34   Edge sorted_edges[G.num_edges] =
35     sort_by_weight(G.edges);
36   Edge mst[G.num_vertices - 1];
37   int mst_idx = 0;
38   for (int i = 0; i < G.num_edges; i++) {
39     Edge e = sorted_edges[i];
40     int u_root = ds.find_set(e.u);
41     int v_root = ds.find_set(e.v);
42     if (u_root != v_root) {
43       mst[mst_idx++] = e;
44       ds.union(u_root, v_root);
45     }
46   }
47   return mst;
48 }
```

Figure 15: Pseudocode for T_I of MST.

in the transcript reflect the intended execution of T_I. However, as we discuss later (§C.4), any order (within a vertex) that respects the weight partial order and that survives the other checks is acceptable, even if such a transcript could not be a result of executing the real T_I.

The test of Disjoint Set semantics is in the helper function `assume_ds_semantics()` (Line 41 in Figure 17). For this test, the prover nondeterminstically reorders H_{pre} as H_{djs} to place all `find_set` operations first (rows with label `FIND-SET` or `UPDATE`).

```
1 Tree mst_efficient(Graph G) {
2   havoc Edge mst[G.num_vertices - 1];
3   havoc Edge not_mst[G.num_edges - G.num_vertices + 1];
4   assume G.edges == mst ∪ not_mst ;
5   AugmentedDisjointSet ds = new AugmentedDisjointSet(G);
6   ds.assume_memory_checks();
7   ds.assume_ds_semantics();
8   for (Edge e : mst) {
9     int u_root = ds.assume_find_set(e.u, e.weight);
10    int v_root = ds.assume_find_set(e.v, e.weight);
11    assume u_root != v_root;
12    ds.assume_union(u_root, v_root, e.weight);
13  }
14  for (Edge e : not_mst) {
15    int u_root = ds.assume_find_set(e.u, e.weight);
16    int v_root = ds.assume_find_set(e.v, e.weight);
17    assume u_root == v_root;
18  }
19  return mst;
20 }
```

Figure 16: Pseudocode for T_E of MST.

Then it performs the check that is detailed in Section 5.5, namely that: for each `FIND-SET` label, new parent must be the root attested to by a sequence of `FIND-SET` and zero or more `UPDATE` operations, where the final row in the sequence has old parent equal to new parent equal to vertex.

C.2 Associating H_{pre} to a valid execution of Kruskal

For efficiency, we would like that creating the needed association does not require the constraints to encode an execution of T_I. To that end, the prover nondeterministically supplies M (the alleged MST), and constraints "check" that this M satisfies the remaining conditions from Section 5.5, namely Conditions 1, 4, and 5.

Condition 1 is that M is a subset of E with cardinality $|V|-1$. This is checked by Line 4 in Figure 16.

To check Conditions 4 and 5 and to finish handling the semantics of `make_set` and `union`, the checker nondeterministically reorders H_{pre} to get H, which has the following structure. Similar to H_{pre}, the first $|V|$ rows of H must have `MAKE-SET` labels, and constraints directly enforce that old parent, new parent, and the vertex are the same (and the weight must be -1) for these entries. The next rows provision for the worst-case number of `UPDATE` operations: they have $2\cdot|E|\cdot\alpha(|V|)$ `UPDATE` labels. These rows are used when enforcing Disjoint Set semantics (§C.1).[2] Then there are $|V|-1$ *three-row groups*; each three-row group consists of a `FIND-SET`, another `FIND-SET`, then a `UNION`, for a total of $3\cdot(|V|-1)$ rows. All remaining rows have `FIND-SET` labels, divided into pairs of rows. These three-row groups and pairs are instrumental in checking Conditions 4 and 5, as explained next.

Recall Condition 4: for each edge e not in M, H reports that the endpoints of e are in the same set. In Kruskal's algorithm, this

[2]The worst-case number of rows is not always needed. However, H_{djs} must act on all rows. We handle this mismatch by allowing `FIND-SET` chains (a `FIND-SET` followed by zero or more `UPDATE`s) to have no-op `UPDATE` entries as padding at the end, which "drains" the pool of excess `UPDATE`s, if needed. These no-op entries have vertex equal to old parent equal to new parent, which passes the checker.

```
struct AugmentedDisjointSet {
  Transcript H; // transcript with rows: (op, vertex, oldP, newP, weight)
  int idx = 0;
  int num_make_set, num_find_set, num_union, num_update, num_ops;
  AugmentedDisjointSet(Graph G) {
    num_make_set = G.num_vertices;
    num_find_set = 2 * G.num_edges;
    num_union = G.num_vertices - 1;
    num_update = 2 * alpha(G.num_vertices) * G.num_edges; // worst-case number of "update"
    num_ops = num_make_set + num_find_set + num_union + num_update;
    H = new Transcript(num_ops);
    for (int i = 0; i < num_make_set; i++) {
      assume H[idx++] == {MAKE-SET, i, i, i, -1};
    }
    for (int i = 0; i < num_update; i++) {
      havoc vertex, oldP, newP, weight;
      assume H[idx++] == {UPDATE, vertex, oldP, newP, weight};
    }
  }
  int assume_find_set(int v, int weight) {
    havoc oldP, newP;
    assume H[idx++] == {FIND-SET, v, oldP, newP, weight};
    return newP;
  }
  void assume_union(int u, int v, int weight) {
    havoc parent, child;
    assume {parent, child} == {u, v} || {parent, child} == {v, u};
    assume H[idx++] == {UNION, child, child, parent, weight};
  }
  void assume_memory_checks() {
    havoc H_par, σ;
    assume H == σ(H_par); // verify that H_par is a permutation of H
    for (int i = 1; i < num_ops; i++) {
      assume H_par[i - 1].vertex <= H_par[i].vertex;
      if (H_par[i - 1].vertex == H_par[i].vertex) {
        assume H_par[i - 1].newP == H_par[i].oldP;
        assume H_par[i - 1].weight <= H_par[i].weight;
      }
    }
  }
  void assume_ds_semantics() {
    havoc H_djs, σ;
    assume H == σ(H_djs); // verify that H_djs is a permutation of H
    assume H_djs[0].op == FIND-SET;
    int i;
    for (i = 1; i < num_find_set + num_update; i++) {
      if (H_djs[i].op == FIND-SET) {
        // H_djs[i-1] is expected to be the last UPDATE in a sequence
        // or a FIND-SET with a 0-length UPDATE sequence.
        // In either case, we need to enforce that the last tuple in the sequence is the root
        assume H_djs[i - 1].vertex == H_djs[i - 1].oldP == H_djs[i - 1].newP;
        assume H_djs[i - 1].weight < H_djs[i].weight;
      } else {
        assume H_djs[i].op == UPDATE;
        assume H_djs[i - 1].oldP == H_djs[i].vertex;
        assume H_djs[i - 1].newP == H_djs[i].newP;
        assume H_djs[i - 1].weight == H_djs[i].weight;
      }
    }
    assume H_djs[i - 1].vertex == H_djs[i - 1].oldP == H_djs[i - 1].newP;
  }
}
```

Figure 17: Pseudocode for `AugmentedDisjointSet` in T_E of MST.

corresponds to two top-level calls to `find_set` (Lines 40 and 41 in Figure 15). For H, this involves assigning each e not in M to a unique `FIND-SET` row-pair, and associating these rows to e as follows:

- Both rows have their weight set to `e.weight`.
- The first row in the pair has a vertex of `e.u`.
- The second row in the pair has a vertex of `e.v`.
- The return values (new parent) of these calls are equal.

Finally, recall Condition 5: for each edge e in M, H reports that the endpoints of e are not in the same set. In Kruskal's algorithm, this corresponds to two top-level calls to `find_set` (Lines 40 and 41 in Figure 15) that return different values and a call to `union` (Line 44 in Figure 15) with their results. For H, this involves assigning each e in M to a three-row group and associating these rows to e as follows.

- All rows have their weight set to `e.weight`.
- The first row in the three-row group has a vertex of `e.u`.
- The second row in the three-row group has a vertex of `e.v`.
- The new parents in these two rows (representing the return values of the `find_set` calls) are not equal.
- These two new parents appear as the old and new parents of the third row (in either order).
- In the final row, the vertex and old parent are equal.

This choice of which root becomes the old or new parent uses the fact that we replaced accesses to `rank` with nondeterministic choice.

None of our checks that H_{pre} is a valid execution of Kruskal's algorithm required actually reasoning about H_{pre}. Instead, they exclusively considered the reorderings H, H_{par}, and H_{djs}. As a final optimization, we remove H_{pre} entirely from T_E, nondeterministically supply H (Line 11 in Figure 17), and then check that H_{par} and H_{djs} are reorderings of H (Lines 32 and 43 respectively in Figure 17). This allows a checker to ensure the existence of a valid H_{pre} without paying the cost of instantiating it in constraints.

C.3 Performance

Avoiding RAM and state machines. Recall from Section 5.5 that the constraint reduction in the compiled T_E, relative to the compiled T_I, is largely driven by avoiding RAM, state machines, and worst-case unrolling. We delve into that point now.

En route to translating T_I to constraints, the contents of the main loop (Lines 38–46 in Figure 15) are first translated to a state machine. To build this state machine, the recursion in `find_set` is translated into two while loops, one which handles the case of following the parent pointer (to find the root) and one which handles the case of re-parenting vertices to point to the root. As a result, each `find_set` call (Lines 40 and 41 in Figure 15) is translated into two complex states with `LOAD`s and `STORE`s on the `parent` array. Another complicated state results from the conditional body on Lines 42–45 in Figure 15: there are `LOAD`s and `STORE`s on the `parent`, `rank`, and `mst` arrays. Furthermore, there are a handful of additional states created by the compiler (resulting from code in between the loops or between the loops and the conditional branch). Because it is not statically known which state the prover will be in at any given time, T_I has to pay for all encoding all states and all transitions, in each of $\Theta(|E| \cdot \alpha(|V|))$ unrolled iterations.

In T_E, this same asymptotic cost appears twice: once as the checks on H_{par} and once as the checks on H_{djs}. Both of these checks involve iterating over a fixed number of rows in the transcript and locally checking properties of adjacent rows. These checks involve conditionals, but the branches are few; crucially, they do not involve `LOAD`s or `STORE`s. As a consequence, the constant on the high-order term (referring to the number of constraints) is far smaller when compiling T_E versus T_I, specifically $52.2\times$ smaller, as stated in Sections 5 and 6.

"Conditional splitting". We also referred to additive improvements when $|E| \gg |V|$, and detail those now. The technique may be of independent interest.

Whereas the constraint encoding of T_I includes logic for untaken conditional branches, the constraint encoding of T_E sheds that overhead. This point is best illustrated by considering the conditional in Line 42 in Figure 15. In an execution of Kruskal, this conditional will be true only $|V|-1$ times (once for each edge added to the MST); yet, if T_I were represented in constraints, those constraints would have to include the logic of the conditional body in all $|E|$ iterations, because the specific iterations in which the conditional is true are not known statically. By contrast, in T_E, two **`assume`** statements (Lines 11 and 17 in Figure 16) presuppose statically that the conditional is true $|V|-1$ times and false $|E|-|V|+1$ times; indeed, the code contains the body of the conditional (Line 12 in Figure 16) only for true iterations, removing the need to pay for it in constraints for false iterations.

One can observe this "conditional splitting" by mapping lines in the body of T_I's main loop (Lines 40–44 in Figure 15) to lines in T_E's two main loops (Lines 9–12 and 15–17 in Figure 16). Conceptually, the constraints are checking the iterations of Kruskal's algorithm out of order, which works because the prover is presumed already to have executed Kruskal.

Because the complexity of our MST is dominated by $O(|E| \cdot \alpha(|V|))$ Disjoint Set operations, the savings of going from $|E|$ `union` instances to $|V|-1$ checks of `UNION` rows does not result in an asymptotic improvement. However, this change does result in significant additive improvements in the number of constraints, particularly when $|E| \gg |V|$.

C.4 Soundness: intuition

Recall our claim from Section C.1: although the prover is free to choose an order in H_{par} that respects the weight partial order, even if the order diverges from what T_I would produce, this freedom is not problematic. Specifically, the prover cannot satisfy the constraints T_E when M is not an MST of G. Ultimately, this claim is established with end-to-end reasoning (§C.5), not by reasoning about all forms of misbehavior. For intuition, however, consider as an example the following attempt by a malicious prover to return something that is not an MST, by using the freedom to reorder rows that have matching `vertex` and `weight` entries in H_{par}.

Suppose the prover tries, for a particular edge e with incident vertices u and v, to maliciously reorder H_{par} to claim `union(u_root, v_root)` occurs strictly before the

`find_set(u)` and `find_set(v)` calls instead of in the order specified by T_I (Lines 40–44 in Figure 15). Let the row corresponding to `union(u_root, v_root)` be:

```
(UNION, u_root, u_root, v_root, weight).
```

Let the row corresponding to the final `UPDATE` required by `find_set(u)` be:

```
(UPDATE, u_root, u_root, u_root, weight).
```

Following the execution of T_I, these rows will be ordered as follows in H_{par}:

```
i  : (UPDATE, u_root, u_root, u_root, weight)
i+1: ( UNION, u_root, u_root, v_root, weight).
```

A nefarious prover might try to reorder these two rows to undo the effect of the `union` call and thus potentially later claim that some other edge is in the MST when it is not. Such a reordering would look like:

```
i  : ( UNION, u_root, u_root, v_root, weight)
i+1: (UPDATE, u_root, u_root, u_root, weight).
```

This behavior is caught by Line 36 in Figure 17 because the `newP` of the first line doesn't match the `oldP` of the second line. The prover could modify the second line to avoid failing this check and produce the following:

```
i  : ( UNION, u_root, u_root, v_root, weight)
i+1: (UPDATE, u_root, v_root, u_root, weight).
```

This passes the check on Line 36 in Figure 17, but now the prover will be caught by various checks on the contents of H_{djs}. The prover can make tweaks to H, H_{par}, and H_{djs} to try to satisfy checks in different parts of the circuit, but ultimately a lying prover will always be caught by at least one of the five checks. This holds for all malicious prover actions, which is established via end-to-end reasoning, summarized in the next section.

C.5 Refinement proof

As usual, we want to show $T_I \leq T_E \leq T_S$, for End-to-end Completeness and End-to-end Soundness (§4.2). We will do this in two pieces. Piece 1 is our usual: $T_I \leq T_E$. For Piece 2, we are trying to establish $T_E \leq T_S$. We will do so by showing $T_E \leq T_I$. Since $T_I \leq T_S$ (indeed, Kruskal's algorithm is known to solve the problem!) we will get the desired refinement relation by transitivity. Notice that our approach here differs from the one taken elsewhere in the paper, where we establish $T_E \leq T_S$ directly; that is because for the other problems, T_E is similar to the respective specification. For the MST problem, however, our T_E is a widget specifically geared to checking the execution of Kruskal's algorithm; indeed, the checks in T_E ensure the existence of an H_{pre} that corresponds to a correct execution of T_I.

We give a high-level overview of the proof based on our discussion of T_I, T_E, and T_S here and in Section 5.5.

We establish that T_I refines transition system T_E by following the approach in Section 4.2 ("Constructing refinement mappings"), in three major steps. First, we instrument T_I to emit the history, H_{pre}, of the Disjoint Set (yielding T_{IE}). Second, we augment this T_{IE} with **`assume`** statements that enforce Conditions 1–5 (to get $\hat{T}_{IE}$) and show that these conditions, when applied over prefixes of H_{pre}, are also loop invariants of the augmented T_I. Finally, we obtain T_E from $\hat{T}_{IE}$ by abstracting away the variables not required by Conditions 1–5 and replacing assignments by **`havoc`** statements. These removed variables include most of the T_I implementation and also the emitted H_{pre}.

We show that T_E refines T_I by following the same general procedure, only this time starting with H and reconstructing H_{pre}. We use H_{pre} and M to reconstruct variables in T_I and then prove that invariants on the Disjoint Set can be reconstructed from these ghost variables (which are effectively prefixes of H_{pre} and M) and from Conditions 1–5. These conditions allow one to reconstruct the state of the Disjoint Set data structure at each point in T_I (and thus all variables in T_I). Additionally, the specifications of operations found in reconstructed states are consistent with return values of operations in H due to Conditions 1–5.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[3] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, 2010.

[4] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61 (12):84–93, 2018.

[5] Sebastian Angel, Andrew J. Blumberg, Eleftherios Ioannidis, and Jess Woods. Efficient representation of numerical optimization problems for SNARKs. In *USENIX Security*, 2022.

[6] Andrew W. Appel. *Program Logics - for Certified Compilers*. Cambridge University Press, 2014.

[7] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1), 1998.

[8] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3), 1998.

[9] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *ACM STOC*, 1991.

[10] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[11] Anindya Banerjee, Ramana Nagasamudram, David A. Naumann, and Mohammad Nikouei. A relational program logic with data abstraction and dynamic framing. *ACM TOPLAS*, jul 2022.

[12] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.

[13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *LFCS*, volume 7734 of *LNCS*, pages 29–43. Springer, 2013.

[14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *FOSAD*, volume 8604 of *LNCS*, pages 146–166. Springer, 2013.

[15] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, August 2014.

[16] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *ITCS*, January 2013.

[17] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *IACR CRYPTO*, 2013.

[18] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.

[19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity, 2019. URL https://ia.cr/2018/046.

[20] Václav E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, 1965.

[21] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25. ACM, 2004.

[22] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In *ACM CCS*, 2020.

[23] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 90–99, 1991.

[24] Tadej Borovšak and Jurij Mihelič. Certifying algorithm for strongly connected components. In *International Electrotechnical and Computer Science Conference*, 09 2016.

[25] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020.

[26] Benjamin Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, December 2012.

[27] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SOSP*, 2013.

[28] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. In *IACR TCC*, 2020. URL https://ia.cr/2020/499.

[29] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *IACR Eurocrypt*, 2020.

[30] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive, Report 2021/651, 2021. URL https://ia.cr/2021/651.

[31] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Security & Privacy*, 2015.

[32] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8(3):174–186, sep 1968.

[33] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[34] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *CAV (2)*, volume 10427 of *LNCS*, pages 126–133. Springer, 2017.

[35] Felix Engelmann, Thomas Kerber, Markulf Kohlweiss, and Mikhail Volkhov. Zswap: zk-SNARK based non-interactive multi-asset swaps. Cryptology ePrint Archive, Paper 2022/1002, 2022. URL https://ia.cr/2022/1002.

[36] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. Zero knowledge static program analysis. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2951–2967. ACM, 2021. doi: 10.1145/3460120.3484795. URL https://doi.org/10.1145/3460120.3484795.

[37] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. In *ESOP*, volume 5502 of *LNCS*, pages 252–266. Springer, 2009.

[38] Cédric Fournet, Chantal Keller, and Vincent Laporte. A certified compiler for verifiable computing. In *Computer Security Foundations Symposium (CSF)*, pages 268–280, 2016.

[39] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In *LICS*, pages 442–451. ACM, 2018.

[40] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over Lagrange-bases for oecumenical non-interactive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. URL https://ia.cr/2019/953.

[41] Nicolas Gailly, Mary Maller, and Anca Nitulescu. SnarkPack: Practical SNARK aggregation. Cryptology ePrint Archive, Report 2021/529, 2021. URL https://ia.cr/2021/529.

[42] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[43] Oded Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1), 2008.

[44] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1), 1989.

[45] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *J. ACM*, 62(4), 2015.

[46] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[47] Jens Groth. On the size of pairing-based non-interactive arguments. In *IACR Eurocrypt*, 2016.

[48] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314. ACM, 2002.

[49] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[50] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: a framework for efficient verifiable computation. In *IEEE S&P*, 2018.

[51] Abhiram Kothapalli, Elisaweta Masserova, and Bryan Parno. Poppins: A direct construction for asymptotically optimal zk-SNARKs. Cryptology ePrint Archive, Report 2020/1318, 2020. URL https://ia.cr/2020/1318.

[52] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021. URL https://ia.cr/2021/370.

[53] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[54] O(1) Labs. Snarky. https://github.com/o1-labs/snarky.

[55] Simon S. Lam and A. Udaya Shankar. Refinement and projection of relational specifications. In *REX Workshop*, volume 430 of *LNCS*, pages 454–486. Springer, 1989.

[56] Leslie Lamport. What it means for a concurrent program to satisfy a specification: Why no one has specified priority. In *POPL*, pages 78–83. ACM Press, 1985.

[57] Leslie Lamport, John Matthews, Mark R. Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. In *ACM SIGOPS European Workshop*, pages 45–48. ACM, 2002.

[58] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/030, 2021. URL https://ia.cr/2021/030.

[59] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[60] Xing Li, Yi Zheng, Kunxian Xia, Tongcheng Sun, and John Beyler. Phantom: An efficient privacy protocol using zk-SNARKs based on smart contracts. Cryptology ePrint Archive, Paper 2020/156, 2020. URL https://ia.cr/2020/156.

[61] libsnark. libsnark. https://github.com/scipr-lab/libsnark.

[62] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151. ACM, 1987.

[63] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM CCS*, 2019.

[64] Monica Marcus and Amir Pnueli. Using ghost variables to prove refinement. In *AMAST*, volume 1101 of *LNCS*, pages 226–240. Springer, 1996.

[65] Robin Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489. William Kaufmann, 1971.

[66] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[67] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3): 287–306, 1987.

[68] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.

[69] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security*, 2020.

[70] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: compiler infrastructure for proof systems, software verification, and more. In *IEEE S&P*, 2022.

[71] Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark W. Barrett. Bounded Verification for Finite-Field-Blasting - In a Compiler for Zero Knowledge Proofs. In *CAV*, volume 13966, pages 154–175. Springer, 2023.

[72] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[73] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

[74] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security & Privacy*, 2013.

[75] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. Formally validating a practical verification condition generator. In *CAV (2)*, volume 12760 of *LNCS*, pages 704–727. Springer, 2021.

[76] Pequin. Pequin: A system for verifying outsourced computations, and applying SNARKs. https://github.com/pepper-project/pequin.

[77] Seth Pettie. Minimum spanning trees. In *Encyclopedia of Algorithms*, pages 1322–1325. Springer, 2016.

[78] J.M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[79] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Security & Privacy*, 2014.

[80] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, 2020.

[81] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[82] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, April 2013.

[83] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*, 2018.

[84] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Conditionally correct superoptimization. In *OOPSLA*, pages 147–162. ACM, 2015.

[85] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *PLDI*, pages 57–69. ACM, 2016.

[86] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[87] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[88] Justin Thaler. Proofs, arguments, and zero-knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2022.

[89] Morgan Thomas. Orbis specification language: a type theory for zk-SNARK programming. Cryptology ePrint Archive, Paper 2022/1003, 2022. URL https://ia.cr/2022/1003.

[90] Rob J. van Glabbeek. The linear time - branching time spectrum I. In *Handbook of Process Algebra*, pages 3–99. North-Holland / Elsevier, 2001.

[91] Riad Wahby. Practical proof systems: implementations, applications, and next steps. https://www.pepper-project.org/simons-vc-survey.pdf, September 2019.

[92] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *ISOC NDSS*, 2015.

[93] Abraham Waksman. A permutation network. *J. ACM*, 15(1): 159–163, jan 1968.

[94] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2), 2015.

[95] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *IEEE S&P*, 2021.

[96] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security*, 2021.

[97] Martin Westerkamp and Jacob Eberhardt. zkRelay: Facilitating sidechains using zkSNARK-based chain-relays. Cryptology ePrint Archive, Paper 2020/433, 2020. URL https://ia.cr/2020/433.

[98] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[99] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security*, 2018.

[100] Hongseok Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.

[101] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM CCS*, 2021. URL https://ia.cr/2021/076.

[102] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE S&P*, 2020.

[103] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 2021.

[104] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE Security & Privacy*, 2018.

[105] ZoKrates. ZoKrates: A toolbox for zkSNARKs on Ethereum. https://github.com/Zokrates/ZoKrates.