ZØ: An Optimizing Distributing Zero-Knowledge Compiler

Matthew Fredrikson
University of Wisconsin

Benjamin Livshits
Microsoft Research

Abstract

Traditionally, confidentiality and integrity have been two desirable design goals that are have been difficult to combine. *Zero-Knowledge Proofs of Knowledge* (ZKPK) offer a rigorous set of cryptographic mechanisms to balance these concerns. However, published uses of ZKPK have been difficult for regular developers to integrate into their code and, on top of that, have not been demonstrated to scale as required by most realistic applications.

This paper presents ZØ (pronounced "zee-not"), a compiler that consumes applications written in C# into code that automatically produces scalable zero-knowledge proofs of knowledge, while automatically splitting applications into distributed multi-tier code. ZØ builds detailed cost models and uses two existing zero-knowledge back-ends with varying performance characteristics to select the most efficient translation. Our case studies have been directly inspired by existing sophisticated widely-deployed commercial products that require both privacy and integrity. The performance delivered by ZØ is as much as 40× faster across six complex applications. We find that when applications are scaled to real-world settings, existing zero-knowledge compilers often produce code that fails to run or even *compile* in a reasonable amount of time. In these cases, ZØ is the only solution that is able to provide an application that works at scale.

1 Introduction

As popular applications rely on personal, privacy-sensitive information about users, factors such as legal regulations, industry self-regulation, and a growing body of privacy-conscious users all pressure developers to respond to demands for privacy. Storing user's data in the cloud creates downsides for the application provider, both immediately and down the road. While policy measures such as DoNotTrack and anonymous advertising identifiers become increasingly popular, a recent trend explored in several research projects has been to move functionality to the *client* [14, 18, 39, 42]. Because execution happens on the client, such as a mobile device or even in the browser, this alone provides a degree of privacy in the computation: only relevant data, if any, is disclosed (to a server). However, in many cases, moving functionality to the client conflicts with a need for computational *integrity*: a malicious client can simply forge the results of a computation.

Traditionally, confidentiality and integrity have been two desirable design goals that are have been difficult to combine. *Zero-Knowledge Proofs of Knowledge* (ZKPK) offer a rigorous set of cryptographic mechanisms to balance these concerns, and recent theoretical developments suggest that they might translate well into practice. In the last several years, zero-knowledge approaches have received a fair bit of attention [24]. The premise of zero-knowledge computation is its promise of both privacy *and* integrity through the mechanism cryptographic proofs. However, published uses of ZKPK [4, 6, 8, 9, 20, 38] have been difficult for regular developers to integrate into their code and, on top of that, have not been demonstrated to scale, as required by most realistic applications.

Zero-knowledge example: pay as you drive insurance: A frequently mentioned application and a good example of where zero-knowledge techniques excel is the practice of *mileage metering* to bill for car insurance: pay as you drive auto insurance is an emerging scheme that involves paying a rate proportional to the number of miles driven, either linearly, or using several billing brackets [5, 40, 43]. Of course, given that the insurance company knows much about the customer, including their address, if daily mileage data is provided, much can be inferred about user's daily activities, where they shop, etc. [16, 31, 32]. The user in this scheme performs a calculation on their own data, but of course the insurance company wants to prevent cheating. Zero-knowledge proofs provide a way to ensure both privacy and integrity, which involves performing the billing computation on the user's hardware (on the *client*), perhaps, monthly, and providing the insurance company with 1) the final bill and 2) a proof of correctness of the accounting calculation, which can be verified by the insurance company (on the *server*) [4, 19, 37, 41].

What we did: In this paper, we present ZØ, a compiler that consumes applications written in a subset of C# into code that produces scalable zero-knowledge proofs of knowledge, while automatically splitting applications into distributed code, to be executed on two (or more)

execution tiers. We are building on very recent developments in zero-knowledge cryptographic techniques [17, 33], exposing to the developer the ability to take advantage of these advances. ZØ builds detailed cost models of the code regions that require ZKPK, and uses existing zero-knowledge back-ends with varying performance characteristics to select the most efficient translation, by formulating and solving constrained numeric optimization problems. Our cost modeling takes advantage of the strengths of both back-ends, while avoiding their weaknesses, both for local and global (distributed) optimization. Using a set of realistic applications that perform tasks such as distributed data mining and crowd-sourced data aggregation, we demonstrate ZØ's ability to produce privacy-preserving code which runs significantly faster than previously possible.

High-level goals: ZØ aims to provide an attractive combination of high-level goals of *privacy*, *integrity*, *expressiveness*, and *performance*. While the first two goals are achieved through the use of zero-knowledge, to support ease of programming and expressiveness, ZØ accepts (a subset of) C#, a widely-used general purpose language as input that can run in many settings. Of course, we are not tied to C# and could support another high-level language such as JavaScript, Java, or C++. Our use of a general-purpose language allows developers to include hundreds or thousands of lines of C# or other .NET code, allowing the construction of full-featured GUI-based distributed applications that support zero-knowledge instead of small examples written in a domain-specific language.

To enable distributed programming wherever .NET code can run, ZØ supports automatic tier-splitting, inspired by distributing compilers such as GWT [21] and Volta [26]. We primarily target client-server computations (two tiers), although other options such as P2P are also supported by ZØ. Code produced by ZØ can be run on desktops, in the cloud, on mobile devices (Windows Phone) and on the web (Silverlight).

Applications: Much of the inspiration for ZØ came from our desire to be able to use ZKPK techniques to build applications directly analogous to some widely-deployed commercial products, as opposed to toy benchmarks. In our studies detailed in Section 6, we show how they can be (re-)built in a privacy- and integrity-preserving way. For example, our FitBit study was inspired by wireless activity tracking devices manufactured by FitBit (`fitbit.com`) and Earndit (`earndit.com`). The Slice study was inspired by purchase tracking software from Slice, Inc. (`slice.com`). The study Waze app was inspired by Waze, a popular crowd-sourced, real-time traffic app for mobile platforms (`waze.com`).

Contributions: We make these contributions:

- This paper proposes ZØ, a distributing compiler that allows developers to create highly performant, large distributed applications, while preserving both privacy and integrity. ZØ uses precisely calibrated *cost models* to choose which underlying zero-knowledge back-end to employ. Based on the cost model, ZØ statically determines the appropriate *splitting perimeter* for the application to achieve best performance and rewrites it to be run on multiple tiers.
- **Developer**: ZØ is designed to be easily accessible to a regular developer; to this end, we expose zero-knowledge functionality via LINQ, language-integrated-queries built into .NET. We demonstrate the expressiveness of the ZØ approach by developing six case studies directly inspired by commercial applications which we hope will become benchmarks for zero-knowledge tools, ranging from personal fitness tracking (Fitbit) to crowd-sourced traffic-based routing (Waze), to personalized shopping scenarios.
- **Cost modeling**: We develop cost models for the individual back-ends, allowing us to perform global cross-tier optimizations. Our cost-fitting models provide an excellent match with the observed performance, with R^2 scores between .98 and .99.
- **Speedup**: We evaluate ZØ on six complex real-life large-scale applications of zero knowledge, focusing on latency and throughput of zero-knowledge tasks. Our global optimizer is fast, completing in under 3 seconds on all programs. ZØ produces code that achieves as much as 40× speedups compared to state-of-the art zero-knowledge systems. We also find that ZØ is able to effectively optimize *across tiers* in a distributed application: while the code it generates may be slower on one tier (we observed one case that was 2× slower for the server), the savings at other tiers are always greater (the same cases, 4× faster on the client).
- **Scale**: At scale, existing zero-knowledge compilers often produce code that fails to run in a reasonable amount of time, or exhaust system resources during compilation. In these cases, ZØ is the *only* solution that is able to provide a working application.

Paper Organization: The rest of the paper is organized as follows. Section 2 provides motivating examples and some background on zero-knowledge. Section 3 gives an overview of the ZØ approach. Section 5 describes the ZØ compiler implementation. Section 4 talks about cost models and both local and global optimizations ZØ performs. Section 5 describes ZØ implementation. Section 6 presents six case studies. Section 7 describes our experimental evaluation. Related work is discussed in Section 8 and Section 9 concludes. The Appendix in

this paper collects additional data and results that we extracted from a longer technical report. We avoided self-citing the technical report to preserve anonymity. Section A presents extra information about our six applications. Section B covers translating LINQ to zero knowledge. Finally, Section C gives some extra details on constrain generation for optimization and other details on our global optimizer.

2 Background

To explain the goals of ZØ concretely, we will demonstrate its functionality on a smartphone application with conflicting privacy and integrity needs.

2.1 Example: Retail Loyalty Card

Figure 1 shows the ZØ code for a personalized retail loyalty card mobile app, with functionality similar to Safeway's "Just for U" application or Walgreens' iOS application. Each time the customer reaches the check-out line, this application interacts with the retail terminal in a bi-directional exchange of information. The exchange takes place using the phone's built-in NFC sensor.

First, the application sends a *discount claim* to the retail terminal, pertaining to the items the customer is about to purchase. This discount is computed based on the customer's previous purchases, using personalization to provide enhanced value and incentive for the customer. Zero-knowledge proofs are supplied to ensure the privacy of the customer's shopping history, without sacrificing the trustworthiness of their discount claim.

Second, the terminal sends a list of purchases to the client, corresponding to the current check-out transaction. This list, along with the customer's other previous purchases, will be stored in a client-side database used to compute a discount the next time the user shops with this retailer.

Application Code: Figure 1 contains C# code for computing the core functionality of this application: using the customer's purchase history to produce a discount, and sending that discount to the retail terminal. It is important to notice that this is standard C#, capable of seamless incorporation into larger bodies of C# code. In fact, ZØ extends the standard C# compiler, and only applies specialized reasoning to classes that inherit from ZØ's `DistributedRuntime` class. All of the UI and external library code can remain in the application, without affecting the performance and functionality of ZØ. This allows ZØ to scale to large applications with arbitrary legacy dependencies, provided that the sections requiring zero-knowledge reasoning are localized and moderate in size. Several important points bear mentioning.

First, of the four functions, two of them, which we call *worker functions*, contain *location annotations*: `GetDiscounts` is constrained to execute on the client

```
1  public class LoyaltyCard : DistributedRuntime
2  {
3    // Local variable declarations
4    [Location(Client)] IEnumerable<int> shophist;
5    [Location(Client)] IEnumerable<int> items;
6    IEnumerable<Triple> automaton;
7    IEnumerable<Pair> transducer;
8
9    public void Initialize(string[] args)
10   {...}
11
12   public void DoWork(string[] args)
13   {
14     var discount =
15       GetDiscounts(shophist, items,
16                    automata, transducer);
17     ApplyDiscount(discount);
18   }
19
20   [Location(Client)]
21   IEnumerable<Pair> GetDiscounts(
22    [MaxSize(Purchases)] IEnumerable<int> history,
23    [MaxSize(Items)] IEnumerable<int> items,
24    [MaxSize(Edges)] IEnumerable<Triple> automata,
25    [MaxSize(States)] IEnumerable<Pair> transducer)
26   {
27     ZeroKnowledgeBegin();
28     // Check that the history is in ascending order
29     var historyAscendingCheck = history.Aggregate(
30       0,
31       (last, curel) => check(last <= curel));
32     // Get the "discount state"
33     var purch_state = history.Aggregate(
34       0,
35       (state, purch) =>
36         automaton.First(
37          trans => (trans.fld(1) == state) &&
38                   (trans.fld(2) == purch)).
39                                    fld(3));
40     var discount = history.Aggregate(
41       new Pair(purch_state, 0),
42       (state, purch) =>
43         new Pair(
44          // Get the next automata state
45          automata.First(
46           trans => (trans.fld(1) == state.fld(1))
47                    && (trans.fld(2) == purch)).
48                    fld(3),
49          // Total the current state discount
50          state.fld(2) + transducer.First(
51            edge => edge.fld(1) == state.fld(1)));
52     ZeroKnowledgeEnd();
53
54     return new IEnumerable<Pair>(discount);
55   }
56
57   [Location(External)] void ApplyDiscount(...)
58   {...}
59 }
```

Figure 1: Running example application: a personalized retail loyalty card.

(e.g., the user's smartphone), and `ApplyDiscount` to `External` (e.g., the retail terminal). ZØ generates separate object code for each of these locations, and inserts code to handle the network transfer and data marshalling for any dependencies between these two functions. In order to streamline the code generated by ZØ, the worker functions must always return `void` or `IEnumerable` objects, which ZØ's underlying runtime is optimized to quickly marshall and transfer.

Second, the target functionality is computed from the

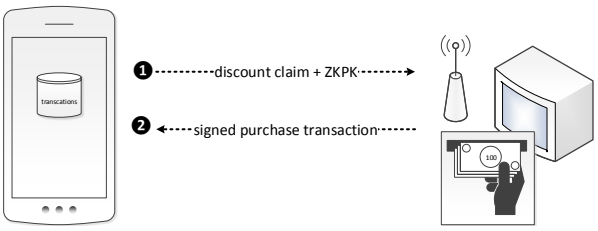


Figure 2: Personalized loyalty card application.

main function `DoWork`, which is called after `Initialize`. `Initialize` gives the application an opportunity to prepare the class's local state by reading sensors, buffering data, etc., and can contain arbitrary C# code. `DoWork` is more constrained: it can contain a sequence of calls to worker functions, with no intermediate local computations, branching statements, or loop statements. This allows ZØ to efficiently compute the dependencies between different tiers. In this case, ZØ determines that the return value of `GetDiscounts` (computed on the smartphone) is always used by `ApplyDiscount` (computed on the retail terminal), and inserts code to package and send, or receive and unpack, the necessary data as well as any accompanying zero-knowledge proofs.

Third, the main code is located in `GetDiscounts`, which takes a list of the user's previous purchases (`history`), the user's current check-out items (`items`), and a finite-state transducer (`automata` and `transducer`), and produces a discount dollar value for transfer to the retail terminal. The transducer is produced by the retailer, and is designed to associate past purchases to items that the customer may be interested in buying in the future; the details of designing the transducer are beyond the scope of this work. `GetDiscounts` begins by checking that the purchases are given in ascending order, by their ID numbers; this is a simple optimization that allows the retailer to minimize the size of the transducer. This check is performed using LINQ's `Aggregate` operator, and ZØ's `check` function, which behaves like an assertion. It then proceeds to traverse the transducer's finite-state machine using the customer's shopping history, effectively loading the history into the transducer's memory in preparation for emitting discount values.

Finally, the customer's current items are processed by traversing the finite-state machine, starting in the final state of the previous traversal, and summing the output of the transducer relation. The final sum is returned to `DoWork` as a discount claim.

Zero-knowledge: The entirety of `GetDiscounts` is computed in zero-knowledge, as indicated by the `ZeroKnoweldgeBegin`() and `ZeroKnowledgeEnd`() annotations. Notice that each statement of this method consists of a LINQ query, giving the computation an overall functional form, without using language features such as references, loops, or conditionals. This is necessary to accommodate faithful translation into code that produces zero-knowledge proofs using the zero-knowledge back-ends discussed in Section 2.2. However, the programmer is still able to express computations in this fragment of standard C#, without dealing with the overhead of inter-language binding between the engines and the main program, and without needing to learn the different input languages understood by each engine.

Finally, a few subtle details of this code bear mentioning. Two of the class variable declarations, `shophist` and `items`, have location annotations that tell ZØ that they should not leave the customer's smartphone without first being processed by zero-knowledge code. This gives the programmer an extra degree of assurance of the code's privacy properties, letting her treat the zero-knowledge code regions like *declassifiers* with additional integrity guarantees. Finally, notice that the parameters to `GetDiscounts` contain `MaxSize` attribute annotations. These optional size annotations allow the ZØ compiler to do precise cost modeling, as explained in Section 4.

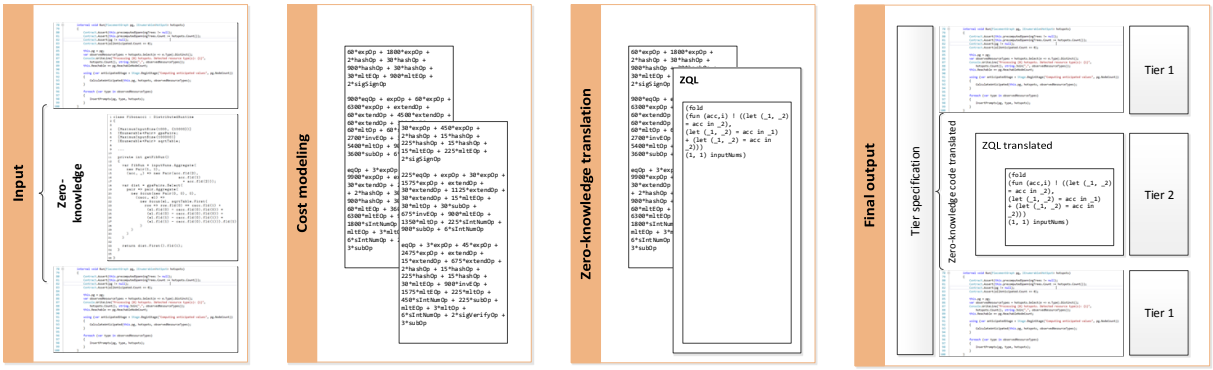
2.2 Zero-Knowledge Back-ends

ZØ relies on two zero-knowledge back-ends, Pinocchio [33] and ZQL [17], to produce code that balances privacy and integrity. Each of these back-ends takes an expression, in the form of executable code in a high-level source language, and produces object code that computes the expression over dynamically-provided inputs while building zero-knowledge proofs for the expression on the given input. These engines have very different characteristics that affect performance and usability in different ways, which we outline here.

Pinocchio: Pinocchio utilizes a novel underlying computation model, *Quadratic Arithmetic Polynomials*, to evaluate an expression and produce zero-knowledge proofs [33]. For some computations, it yields performance gains several orders of magnitude beyond previous systems that gave similar functionality, producing proofs of a *constant size* regardless of the size or structure of the target expression.

The expression language supported by Pinocchio is a strict subset of C, and the object created for evaluation is an *arithmetic circuit* [33]. The fact that the target circuit must be finite, and cannot encode *side-effects*, imposes necessary conditions on the parts of C that are available. Loops and conditionals are "unrolled" during compilation, so all loops must have static bounds. Likewise, pointers and array indices must be compile-time constants, or simple loop variables (as these are unrolled), thus simplifying cost modeling. For this paper we used a publicly released version of Pinoccio 0.4 obtained from the public distribution[1].

[1] `https://vc.codeplex.com/downloads/get/714129`



1) Input is supplied as C# code, containing a mix of ZK blocks are regular blocks.

2) Cost modes for ZQL and Pinoccio are used to decide ZK runtime costs.

3) Appropriate ZK translation are generated in .NET IL.

4) Final .NET DLLs are produced for each tier

Figure 3: ZØ architecture.

ZQL: ZQL utilizes several fairly recent advances in the theory of zero-knowledge proofs to produce efficient verified private code that operates over functional lists [17]. The underlying cryptographic machinery used by ZQL is more traditional than that of Pinocchio, relying heavily on homomorphic commitment schemes to provide its guarantees. The expression language supported by ZQL is a simple functional language without side effects, and limited operator support. In a nutshell, ZQL supports map and fold operations, as well as find operations over tuples of integers. Boolean expressions can only be used inside of find operations, and are currently limited to conjunctions of equality tests; all forms of inequality are not explicitly supported, although the authors plan to support these operations in future versions. In terms of arithmetic, addition, subtraction, and multiplication are supported. Finally, multiple operations can be sequenced using classic functional let bindings. Although these constructs might seem modest at first blush, the ability to perform table lookups using find allows for the evaluation of logic gates, and the list-based map and fold operations place no upper-bound on the size of the program's input, as in the case of Pinocchio. We obtained a version of ZQL from its authors.

3 Overview

Figure 3 shows the architecture of the ZØ compiler. The developer provides as input a set of C# source files, which may include arbitrary regions of legacy and library code as well as functionality targeted towards zero-knowledge proof generation. ZØ then enters a *cost modeling* stage, analyzing the zero-knowledge regions, building performance models that characterize the cost of providing zero-knowledge proof generatiion and verification code for each available zero-knowledge back-end. These models take the form of polynomials over the size of the input data to the zero-knowledge region in the original C# application. ZØ then compares the models to determine which engine the application should use for each C# statement in the region, and translates the C# code (depicted in the *zero-knowledge translation* stage of Figure 3) into expressions understood by the appropriate zero-knowledge engine. In the *final output* stage (Figure 3), ZØ decides how to split the application across tiers to maximize performance, given privacy annotations as well as relative *costs* for transmitting data and computing at each tier.

This translation yields a separate module which is callable from the original application, either as an *arithmetic circuit* (Pinocchio) or standard .NET bytecode (ZQL). Finally, ZØ partitions the original C# code, along with the zero-knowledge modules compiled in the previous step, into multiple applications to run at each *service tier*. During partitioning, ZØ inserts code to perform communication, synchronization, data marshaling, and zero-knowledge proof transfer in parallel to the original application code. The resulting modules are standard .NET bytecode that can be run on the proper tiers without the need for additional specialized software.

Optimization & cost models: Even apparently straightforward applications like the personalized loyalty card app discussed in Section 2.1 contain subtle characteristics that might make zero-knowledge proof generation expensive. It is often the case that one zero-knowledge engine offers significantly better performance for a particular statement, and selecting the appropriate engine for each computation in the zero-knowledge region means the difference between a scalable, low-latency implementation and one that requires hours or days to execute.

For the loyalty card application in Figure 1, it turns out that the inequality comparisons are better handled by Pinocchio, whereas the table lookups needed to execute the transducer are very inexpensive when performed by ZQL. A comparison of the times to perform the opera-

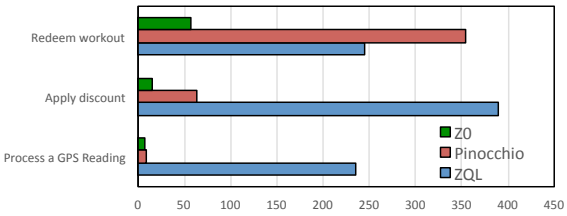


Figure 4: Comparison of times for several applications.

tion on the *y*-axis for several applications from Section 6 is shown in Figure 4. We can see dramatic differences in performance between the back-ends, with the ZØ approach out-performing either of the two back-ends. ZØ addresses these performance differences by building detailed performance models for each statement in the zero-knowledge region.

Distributed configuration: To support a variety of distributed scenarios, ZØ allows the developer to place code on several different tiers, which are specified using the following *tier labels*: Client (end-user's primary device), External (provider's servers), ClientShare (peer-to-peer nodes), and ClientResource (additional hosts owned by end-user). Tiers impose data confidentiality and integrity constraints, as ZØ makes assumptions about the *trust relationships* between tiers.

The figure in this paragraph shows these relationships; white cells indicate trust, and gray the opposite. At compile time, the user can modify the configuration by specifying *weights* on each tier label indicating the relative cost of computation at that tier, as well as the cost of communication between tiers. ZØ uses these weights during optimization to determine the best placement of code and data amongst the tiers. Data privacy constraints are given by the programmer by marking certain variables as *private* to a particular tier using the attribute [Private(T_L)], where T_L specifies the tier to which the data is considered private (e.g., Client, External, ...).

	C	CS	CR	E
C				
CS				
CR				
E				

Note that by design, these annotations are lightweight: they are only needed on (the few) variables that must be kept confidential. Most can be declared without any annotations at all.

When ZØ compiles the application and runs a global optimization described in Section 4.2 to place each worker method on a specific tier, privacy annotations are used in part to determine on which tiers a method may reside. These constraints are *hard*, meaning that a privacy annotation that requires a less performant compilation configuration will always be respected; if the privacy constraints conflict with each other, then compilation will not terminate early. Privacy annotations are propagated transitively using a local dataflow analysis, so that dependent variables have matching annotations.

Threat model: Because of its reliance on zero-knowledge back-ends, ZØ makes all of the assumptions needed for security by ZQL [17] and Pinocchio [33]. The result of ZØ compilation will be executed on one or more tiers. Privacy is violated when the trust relationships given in the previous section are violated. We assume that tiers cannot learn information by means other than direct communication, i.e. Server cannot obtain the list of purchases through side channels, for instance, unless it is directly shared by Client. Our applications that use secret sharing (Waze and Slice in Section 6) also assume that P2P clients do not collude.

4 Cost Models & Optimizations

This section discusses ZØ's cost modeling approach to optimizing zero-knowledge computations. As outlined in Section 3, in many cases one zero-knowledge engine will outperform the other on a particular computation by a significant factor, giving ZØ a key opportunity to optimize the code it produces. ZØ optimizes zero-knowledge regions by building detailed performance models that characterize the cost of building and verifying zero-knowledge proofs in each engine. We are able to accomplish this with reasonable accuracy because the execution depth of zero-knowledge regions is statically-bounded (a necessary condition imposed by the underlying engines), and the evaluation of zero-knowledge code universally relies on a few primitive operations. This allows ZØ to build static *cost models* as polynomials over the number of primitive operations each region must execute.

Section 4.1 discusses local optimizations within a given zero-knowledge region to decide which back-end to use. Section 4.2 proposes a split for the entire application designed for maximal performance.

4.1 Local Optimization

In order to build cost models for ZQL code, we execute the F# "object code" generated by ZQL's compiler *symbolically*. Symbolic data is represented by polynomials that characterize the size of the corresponding concrete data, or structured sets of polynomials in the case of structured data types. The symbolic operation for each ZQL operation accumulates terms on a polynomial that characterize the cost of that operation in terms of the size of its input data, and returns a new polynomial that characterizes the cost of producing of the result. Because the execution depth of iteration commands is always a polynomial function of the size of the inputs, and ZQL programs do not contain branching, accumulating a cost polynomial by symbolic execution necessarily accounts for *all* of the operations contained in a ZQL program.

Recall that Pinocchio compiles C code into a circuit,

	ZQL			Pinocchio		
	Setup	**Prover**	**Verif.**	**Keygen**	**Prover**	**Verif.**
FitBit	0.01	1.81	0.10	0.39	0.20	0.00
Waze	0.11	0.29	0.25	0.04	0.02	0.00
Loyalty	0.03	0.35	0.11	0.31	0.20	0.00
Slice	0.06	0.41	0.32	0.05	0.03	0.00
Average	0.05	0.72	0.20	0.20	0.11	0.00

Figure 5: Absolute regression error (in seconds) for selected applications.

which is evaluated by a specialized runtime to produce and verify zero-knowledge proofs. The Pinocchio runtime executes roughly the same code to evaluate every circuit, varying only on the number of times each operation is executed to handle every element of each input list and every operation in the circuit. We build a set of static polynomials that characterize the execution time of the runtime in terms of the size of the input circuit, i.e., the number of I/O wires and multiplication gates it contains. For example, the cost of the verification stage is given by the polynomial:

$$ExpMulB \times NInputs + 12 \times Pair + VerifyConst$$

In this polynomial, *ExpMulB* corresponds to the amount of time taken to complete a multi-Exponentiation on the Pinocchio's base elliptic curve, *NInputs* to the number of input wires in the circuit, *Pair* to the field pairing cost [33], and *VerifyConst* to a fixed setup cost for the verification stage. Similar polynomials are derived for the key generation and computation stages of Pinocchio's runtime.

We use least-squares regression to derive coefficients for all models except those for Pinocchio's compute-stage model, which contains a non-linear term corresponding to the $O(n \cdot \log^2 n)$ runtime of polynomial interpolation. To cope with the non-linearity in Pinocchio's compute-stage model, we use the Gauss-Newton method [35] with at most 1,000 iterations and a randomly-chosen starting point.

Cost-fitting results: To derive the necessary coefficients for our models, we built a regression training application in ZØ consisting of several basic operations likely to appear in zero-knowledge applications. The training application takes as input a list of integers, and computes an aggregate sum, scalar product, second-degree polynomial, boolean mapping, and table lookup on the list. We compiled this application to use both all-ZQL and all-Pinocchio zero knowledge computations, and ran it ten times for each zero-knowledge engine using a fixed list size ($n = 100$). We performed regression to learn coefficients corresponding to the execution time of each primitive operation appearing in the cost model. We then compiled a representative subset of the applications described in Section 6 to use either all-ZQL or all-Pinocchio zero-knowledge computations, executed each zero-knowledge region ten times, and recorded the deviation between execution time predicted by the regression-trained cost models and the mean execution time observed over all experiments for a given application. Figure 5 presents the prediction error of the trained cost models in terms of the total zero-knowledge execution time in seconds. Note that the models derived for Pinocchio are generally more accurate in terms of relative error than those for ZQL, but the error in both cases is quite small: the greatest Pinocchio error is 0.39 seconds (on FitBit's key generation routine), while the greatest ZQL error is 1.81 seconds (on FitBit's prover routine). The coefficient of determination (R^2) for each performance model is at least 0.98, indicating a precise fit of the models to the execution time.

Summary: To summarize, ZØ is able to build performance models of zero-knowledge regions that predict actual execution time within tenths of a second in most cases, which provides ample accuracy to make a correct decision when selecting zero-knowledge engines at compile-time.

4.2 Global Optimization

ZØ builds cost polynomials to characterize the expense of each zero-knowledge operation in the target application. However, selecting the least expensive engine for each operation is oftentimes not as straightforward as evaluating each polynomial at a target input size and choosing the engine corresponding to the lesser value — it may be the case that a less expensive operation on the prover's side requires a more expensive operation on the verifier's side, and depending on the application computation may be more expensive for the verifier. Alternatively, there may be several ways to partition an application between tiers while preserving the privacy of variables at each tier, with each partition yielding a different trade-off between computation and communication cost. To address these concerns, ZØ performs *global optimization* on the application to balance the cost of computation and communication among differentiated tiers. More details of the global optimization engine are given in Appendix C.

Performance of global optimization: We implemented our global optimization algorithm as part of the ZØ compiler. We use CCI2 to traverse the AST of the target code, and our cost modeler to generate the objective function.

To perform the constrained optimization needed to find an optimal solution, we used the Nelder-Mead method [35] with at most 100 iterations. We looked for integer solutions over the full space of possible tier splittings.

The results are presented in Figure 6. Each application resulted in between 30 and 300 con-

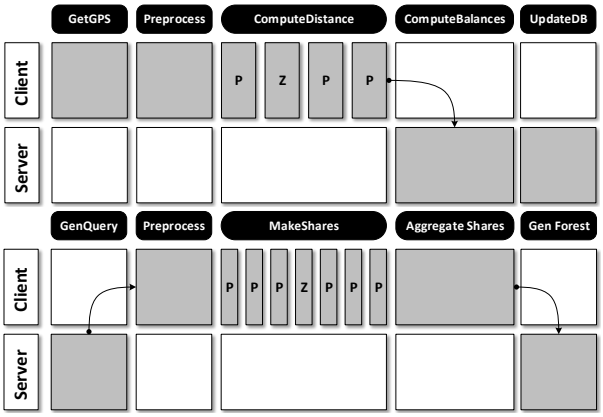


Figure 7: Splits produced by global ZØ optimizations, for FitBit and Slice. For each phase of the computation, grey cells indicate computation location (or tier) chosen by the optimizer, with **P** and **Z** denoting ZQL and Pinocchio back-ends, respectively.

straints, and the constraint solver found an optimal solution in under three seconds for all applications. Because Nelder-Mead is an approximate numerical optimization algorithm, it is possible that it would return a *local* minimum.

However, we checked the solution returned for each application, and verified that it corresponded to the true global minimum. Figure 7 shows examples of ZØ-computed global splits for two representative applications.

	Constr.	Time
FitBit	179	1.50
Loyalty	38	0.01
Waze	263	2.65
Sice	230	2.14

Figure 6: Global optimization performance, showing solver time in seconds for the benchmarks in Section 6.

Main program definition

Program	::=	InitBlock MainBlock MethodDef* TypeDef*
InitBlock	::=	CSMethodSig VarDecl*
MainBlock	::=	CSMethodSig WorkerStmt+
MethodDef	::=	CSMethodSig (ExternCall | LinqStmt)+
TypeDef	::=	`class` Id { CSFieldDef + }
CSMethodSig	::=	PrivacyAnnot CSType Id(...){ ... }

Statements

WorkerStmt	::=	SleepStmt | CallStmt | ZKAnnot
SleepStmt	::=	`WorkerSleep`(Integer, Integer, Integer)
CallStmt	::=	(Id =)? MethodCall
ExternCall	::=	`return External.`Id"("Id*")"
LinqStmt	::=	(Id =)? LinqExpr
VarDecl	::=	(PrivacyAnnot | SizeAnnot)? Id(= CSExpr)?

Expressions

Lambda	::=	"("Id*")" ⇒ LambdaExpr
LambdaExpr	::=	MethodCall | ArithOrBoolExpr | FieldExpr | NewObj
LinqExpr	::=	LambdaLinqExpr | ZipLinqExpr
LambdaLinqExpr	::=	Id.LambdaLinqId(Lambda)
LambdaLinqId	::=	Select | Aggregate | First
ZipLinqExpr	::=	Id.`Zip`(Id, NewAnonObj)
MethodCall	::=	Id "("LambdaExpr*")"
NewObj	::=	NewAnonObj | NewStaticObj
NewAnonObj	::=	`new` {(Id = LambdaExpr)+}
NewStaticObj	::=	`new` MethodCall
FieldExpr	::=	Id.`fld`⟨Type⟩(Int)

Annotations

ZKAnnotat	::=	ZeroKnowledgeBegin() | ZeroKnowledgeEnd()
PrivacyAnnot	::=	[Private(T_L)]
SizeAnnot	::=	[MaximumInputSize(Int+)]

Figure 8: BNF syntax for the subset of C# supported by ZØ. Entities prefixed with CS correspond to the corresponding C# syntax entity.

5 Implementation

In order to make privacy analysis, zero-knowledge translation, and aggressive optimization feasible for the programmer, ZØ supports a subset of C# that includes certain LINQ (language integrated queries [36]) functionality and support for external code. To ensure that the external code does not interfere with the privacy, integrity, and optimization goals of ZØ, the contexts in which it is allowed are limited in some cases. The syntax accepted by ZØ is summarized in Figure 8.

The main program is structured into three parts: an initialization routine (InitBlock, contained in a method `Initialize`), the main body (MainBlock, contained in a method `DoWork`), and the worker methods (MethodDef). The initialization routine may consist of a sequence of arbitrary C# assignment statements, including calls to methods in external libraries not written in ZØ's input language. The main block consists of a sequence of method calls, assignment statements, and sleep statements. Each method call in the main body must be to a worker method defined in the ZØ application.

Zero-knowledge regions: The body of each worker method can contain calls to external methods, standard C# arithmetic and Boolean operations, and a subset of the standard LINQ data processing operations. Regions comprised of LINQ operations can be converted into zero-knowledge proof-generating object code using either available zero-knowledge engine (ZQL or Pinocchio). The supported LINQ operations include Select, Aggregate, First, and Zip. Select provides the ability to project the data in one list into a new list, while performing arithmetic and Boolean operations on each item in the original source list. Aggregate provides the ability to compute iterated functions over a list, maintaining an order-sensitive state through the iteration, which is eventually returned as the result of the operation. First provides the ability to perform searches over lists, using a programmer-defined predicate to determine which element of the list to match. Finally, Zip provides the ability to combine multiple lists, applying arithmetic and Boolean operations to each pair of items from the original source lists.

Zero-knowledge regions are specified by the programmer using a pair of methods ZeroKnowledgeBegin and ZeroKnowledgeEnd. Because zero-knowledge computations

provide both integrity and privacy, these annotations serve a dual purpose. First, the programmer is denoting that the variables which are *live* [1] at the end of a zero-knowledge region are trusted across all tiers: the values have accompanying proofs that any tier can examine to verify that the computations in the zero-knowledge region are performed correctly. Second, these regions serve to *declassify* private values that are used as inputs to a zero-knowledge region; this is in line with the approach taken by ZQL [17]. Because the inputs to zero-knowledge regions are kept private, except in cases where the computations are in some way invertible, the output values that depend on these inputs are considered public to all tiers.

Formal reasoning about composing proofs obtained from different zero-knowledge back-ends remains an avenue for future work. Because this work involves experimentation with very recent cryptographic tools, we are not aware of a readily-available composition theorem that would support reasoning about Pinocchio and ZQL.

Code splitting: ZØ partitions the given target application into code that runs on multiple tiers, inserting marshalling and synchronization code [21, 26] as necessary to ensure that the compiled functionality matches that specified in the original input program. The rewrite process is implemented as a bytecode-to-bytecode transformation within the CCI 2 rewriting framework for .NET [29]. We assume that the target tier for each method is provided as input to the compiler by the optimizer, as described in Section 4.2.

Code partitioning between tiers takes place at method granularity, and data partitioning is determined by the chosen code partition; data is transmitted between tiers on-demand, with all of the data represented by a variable used by a particular method being transmitted at once as it becomes available. Only worker methods can be split between different tiers, so all external code referenced by the application is present on each tier. This allows the compiler to avoid a potentially expensive deep-dependency analysis of the referenced external code, while keeping the dependency analysis of the target application localized to `DoWork`. More details are given in Appendix B.

Runtime support: The architectural principle that guides ZØ's tier-splitting algorithm can be summarized as follows: *whenever possible, delegate the data communication and synchronization operations necessary to support functionality to a runtime API.* Each application compiled by ZØ is linked to a runtime library that provides an API for communicating data and synchronization between separate tiers. When the compiler performs tier splitting, rather than inlining complex code to perform the tasks, simple calls to this API are inserted to perform the "heavy lifting" of tier crossings at runtime.

6 Motivating Case Studies

This section presents six case studies. Appendix A presents architectural diagrams and a detailed description of the algorithm for each of these application. Similarly to [17], we assume that the sensor readings devices can are trusted and untampered with, and come signed by their producer, but the machine or mobile phone (Client tier) that performs the distance computation is not. Techniques for building trust deeper into the platform are complimentary to our work [25].

1) Walk for Charity with FitBit: Several programs exist for paying users for the amount of physical exercise they perform, either directly in the form of rewards, or indirectly by making charitable donations on their behalf, such as `earndit.com`. This works by requiring users to log their exercise habits using a FitBit or other sensor device to measure the distance the user walks, runs, or bikes, and send the logs to a centralized server.

Privacy: The user may not want to reveal their *detailed physical activities* or *exercise route* to a relatively untrusted third party.

Integrity: The service is spending money on the basis of distance derived from sensor logs. If the distance computation can be subverted, the possibility for fraud arises, analogously to pay as you drive insurance [5, 40, 43].

Solution: Keep all sensor readings local to the user's machine (laptop or mobile device), perform the distance computation locally, on the client, send the result of the distance computation to the centralized third-party server. Use ZKPK to ensure that the distance computation is performed correctly. This approach is similar to what has been advocated for smart metering [37].

2) Supervised Studies in Social Sciences: Many scientific studies, especially in medical and social sciences, require subjects to wear sensors and undergo protocols that provide information about their physiological and psychological state. A study that seeks to understand the effect of common workplace events on worker's stress levels might require a participant to wear a galvanic skin response sensor and a camera to detect face-to-face interactions.

Privacy: Participants may have concerns about the use of their physiological measurements or, most prominently, the processing of images taken from their cameras.

Integrity: These studies typically involve payment given to subjects. Subjects concerned about their privacy, or those who simply do not want to wear intrusive sensor devices, have an incentive to *fake* the data used in the study.

Solution: Have all sensors associated with the study report readings to the subject's machine (desktop or mobile phone). This machine performs aggregate computations

relevant to the actual study on the readings, reporting results and discarding the raw sensor readings. ZKPK is used to ensure that the readings are processed correctly.

3) Personalized Loyalty Cards: Many of today's large retailers such as Target, BestBuy, etc. use customer loyalty cards to encourage repeat visits. Typically, the customer must enroll in a loyalty program, and receive a card that can be applied to receive discounts in future visits. Recently, certain retailers (e.g., Safeway) have begun personalizing this process by using the customer's past purchase history (available because of the association between checkout and loyalty card) to create discounts available only to one particular customer. Depending on the retailer, these discounts can be sent to the customer's mobile phone, or applied automatically at checkout.

Privacy: Many people are not comfortable with a retailer tracking their purchases. This is most readily illustrated by a recent scandal with Target discovering that a teenage girl was pregnant before her parents did [15].

Integrity: Retailers offer discounts on the basis of past purchase history. If a customer could fake a purchase history, they might be able to obtain a discount for an item of their choosing. Moreover, having a reproducible strategy for "generating" discounts might create a serious problem for the retailer, similar to those experienced by some retailers that were overly generous in offering Groupons [34].

Solution: The solution is discussed in Section 2.1.

4) Crowd-sourced Traffic Statistics: Several mobile applications such as Waze (`waze.com`) and Google Maps provide traffic congestion information to end-users based on the combined GPS readings of everyone using the app.

Privacy: Users do not want to share their location with the app's servers, or the general public (in the case of a distributed protocol).

Integrity: The app needs reliable GPS readings from users to provide its core functionality. If users wish to "game" the system by providing fake GPS readings while receiving the end-product, the integrity of traffic data is compromised for everyone.

Solution: Let the users keep their GPS readings local, and take part in a distributed protocol to compute local density information for transmission to the app's central server. Clients represent their location on a map using a vector, represented as a set of secret shares, which can be added to the other clients' vector shares to derive the overall traffic density map. When each client sends their summed shares to the server, it can reconstruct the density map by combining the shares, as detailed in the appendix.

5) CNIDS: Collaborative intrusion detection (CNIDS) has long been a goal of security practitioners [27]. In the CNIDS scenario, multiple (distrustful) organizations share the results of their network intrusion detection sensors, to provide their peers with advanced warning about possible threats. A practical approach involves sharing IP blacklists: when an IP generates a valid NIDS alert on one organization's network, the IP is recorded and sent to the other participating organizations.

Privacy: NIDS operate on highly sensitive data — raw network traces. Organizations participating in CNIDS do not want to share their traces with other organizations, and in many cases, may be prohibited from doing so by law or organizational policy.

Integrity: Given the privacy concern and the benefits of participating, some organizations may want to freeload by suppressing their own NIDS alerts. Additionally, if an adversary manages to compromise a participating network, it may choose to suppress or even generate false alerts, which may result in a denial of service for the targeted IP address.

Solution: Provide a ZKPK for the NIDS signature-matching process, to prove that a claimed intrusion is correct according to the signature. Note that this approach assumes that raw network data coming into the NIDS has not been tampered with, but that the machine performing the signature matching may not be trusted.

6) Slice: Organizing Shopping: Slice (`slice.com`) is a service that takes as input a user's past purchase history from their email mailbox, and provides various services using that data. One such service is product recommendation — given everybody's past purchase history, slice can build classifiers that predict a likely "next" purchase.

Privacy: Handing one's entire purchase history to a profit-driven third party has obvious privacy implications. So does the troubling need to share one's email credentials with Slice at the moment.

Integrity: A user, particularly one concerned about privacy, might provide fake data to Slice in order to obtain the useful classifier, which would pollute Slice's data for everyone and jeapordize Slice's ability to profit from the classifier.

Solution: Keep the user's purchase history local, and have the users take part in a distributed protocol in order to produce the classifier for Slice. Use ZKPK to ensure that no user is able to subvert the distributed classifier computation.

7 Experimental Evaluation

All experiments were performed on a Windows Server 2012 R2 machine with two 3.0 GHz 64-bit cores with 8 GB of RAM. All reported timing measurements correspond only to the zero-knowledge portion

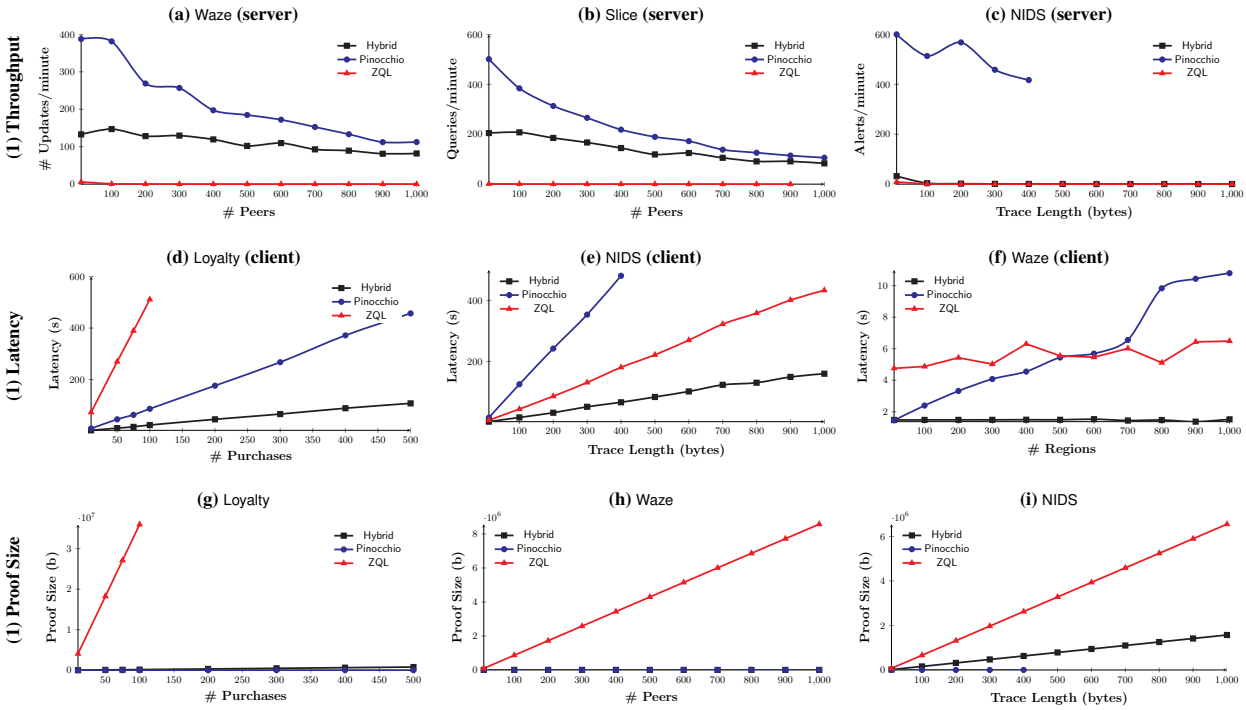


Figure 9: (1) Throughput, (2) latency, and (3) proof size for a characteristic sample of application functionality.

Scaling	ZØ scales to all application configurations. Others may time out, or not even compile in fewer than 20 minutes, on some parameter settings: 100-byte traces (NIDS), >100 peers (Slice), large automata (2000 edges) (Loyalty).
Latency	ZØ improves up to 40×, ≈ 5–13× on average
Proof size	ZØ almost always less than 1 MB, at most 1.5 MB. ZQL proofs can be tens or hundreds of MBs.
Global tradeoffs	ZØ may be slower at one tier (2× slower for Waze server), but savings at other tiers is always much greater (4× faster for Waze clients)

Figure 10: Performance summary.

	Pinocchio Speedup		ZQL Speedup	
	Average	Max	Average	Max
FitBit	6.4	6.6	4.5	4.7
Study	1.0	1.0	39.7	40.3
Loyalty	4.1	4.2	10.1	21.8
Waze	4.0	7.1	4.3	4.7
CNIDS	5.3	7.3	2.7	2.7
Slice	2.5	4.1	8.1	12.9
Average	**4.5**		**13.0**	

Figure 11: Latency speedup factors for each application.

of the application's execution time, as this is the only portion that our compiler attempts to optimize.

The execution time of the ZK code is generally much higher that of the rest of the application, so focusing on these parts gives an accurate picture of the overall execution time. Each zero-knowledge proof generation and verification task was terminated after ten minutes. Our implementation uses 1,024-bit RSA keys for ZQL computations. Integers in Pinocchio circuits were configured to have 32-bits for comparison operations, and operate over a 245-bit field.

Figure 10 summarizes the key performance results from our experiments. We found that the ZØ-generated code gave significant performance benefits both in terms of computation time and proof size: up to 40× runtime speedup, with most proofs below 1 MB (the largest being ≈ 1.9 MB). Furthermore, we saw that global optimization is necessary to arrive at an ideal performance profile: some applications perform noticeably worse at one tier, but in each case the speedup at another tier was always greater. For example, the code ZØ generated for the Waze server ran ≈ 2× slower than Pinocchio's on average, but the client tier experienced ≈ 4× reduced latency.

Figure 11 shows the latency speedups across all applications. The average speedup delivered by ZØ is 4.5× compared to Pinocchio and 13× compared to ZQL.

Results: Space limitations do not allow us to present our measurements exhaustively. Instead, Figure 9 shows a sample of the runtime characteristics for our target applications. Rather than giving raw execution times, the results are broken into three categories: *throughput*, *latency*, and *proof size*. These metrics were selected to

more clearly depict the impact of zero-knowledge techniques on each application.

Throughput: Figure 9(a)–(c) shows the results of three experiments involving throughput. Figure 9(a) shows the server's throughput for the Waze application, which corresponds to the number location updates per minute the server can handle as the number of users (n) increases. Notice that Pinocchio outpaces both the hybrid and ZQL compilations by about 2× on average. This is a result of the global optimization engine: verification in Pinocchio is very fast, whereas the time to construct a proof can be quite slow: in this case, the proof construction phase was up to 7× slower than the hybrid solution. This is critical, *as proof construction takes place on the client where resources are especially constrained for the application.* The discrepancy in resources is correctly used by ZØ to optimize for a lighter client workload at the expense of greater server overhead.

Figure 9(b) shows the number of random forest construction queries per minute the Slice server is able to handle, as the number of participating peers increases. As with Waze the Pinocchio solution dominates the ZØ solution at all data points because of the greater expensive of constructing proofs on the client, where the Pinocchio solution is up to 4× slower than the ZØ solution.

Figure 9(c) shows the number of intrusion alerts per minute the collaborative NIDS server can handle as the number of bytes in the intrusion trace increases. Notice that Pinocchio outperforms at a few small data points, but fails to scale to any larger points. This is not because the server-side component is unable to scale, but rather the client timed out at these settings. For the remaining points, the ZØ solution outperforms the others by about 4×, and is the only solution that is able to scale to even the modest intrusion trace length of 1 KB.

Latency: Figure 9(d)–(f) shows the results of three experiments involving latency. Latency is always measured in seconds, and has a uniform upper bound of 600 seconds, which corresponds to our experimental timeout. Figure 9(d) shows the latency of the client side of the Loyalty application as the number of purchases used to personalize discounts (n) increases. The ZØ solution far outpaces both alternatives at all data points (4–22× improvement). These experiments were performed for an automaton with about 75 edges. We found that when we scaled the automaton to more realistic sizes (a few thousand edges), the ZØ solution was the only one capable of completing *any* number of purchases before timing out, and the Pinocchio compiler timed out after 20 minutes. For longer purchase histories, the ZØ solution completes in just over 1.5 minutes, which is ample time if the application is location-aware and begins proving a set of discounts when the user enters the store.

Figure 9(e) shows the NIDS client's latency to demonstrate that a single intrusion is present in a trace. Pinocchio times out at all points beyond 300 bytes, whereas ZØ is about 2.7× faster than ZQL. Otherwise, we see that as long as intrusions are spaced more than two-and-a-half minutes (159 seconds) apart, the NIDS client has enough time to build proofs for each intrusion trace.

Figure 9(f) shows the latency of the Waze client to send traffic statistics for a single location query as the size of the map (n) increases. First notice that the ZØ solution is essentially constant, not varying by more than 1.5 seconds between any two data points. The other solutions require as much as 4–7× as long to process a query on the client, which will limit the quality (i.e., recency) of the statistics the server is able to gather over time. Second, notice that at about $n = 700$, ZQL becomes more performant than Pinocchio. This is because as the map increases, the size of the lookup table needed to encode the regions increases. Pinocchio is not able to perform lookups as quickly as ZQL, so the portion of the computation needed for lookups becomes more significant at higher values of n. ZQL performs worse at lower values because most of the computation corresponds to the multiplications needed to compute secret shares, which it does not complete as quickly as Pinocchio.

Proof Size: Figure 9(g)–(i) shows the results of experiments involving the size of the zero-knowledge proof in various applications. We always measure in bytes, and do not display a curve for the Pinocchio solutions, as it is constant across input size and is usually too small to distinguish on the same scale as the ZQL and ZØ solutions. Figure 9(g) shows the proof size for the Loyalty application as the number of past purchases (n) varies. While the Pinocchio solution of course dominates the others by this metric (864 bytes), as we know from previous experiments (Figure 9(d)) it does not scale in terms of Latency. The ZØ proof size remains nearly constant, always under 500 KB, whereas the ZQL solution requires at least three megabytes (to perform the inequality checks at the beginning), and finishes at about 100 megabytes. Note that we obtained the point at $n = 300$ despite the timeout, by letting the prover run for longer in this single instance. Because the Loyalty application needs to communicate this proof wirelessly to a POS terminal, size is crucial, and the ZØ solution offers the best overall characteristics in terms of size and latency.

Figure 9(h) shows the proof size for the Waze application as the number of peers varies. Again, Pinocchio dominates (2 KB), but the tradeoff in latency for this proof size is quite high (Figure 9(f)). The ZØ proof size remains constant at around 5 KB because the only processing done by ZQL is table lookups, which have a constant proof size. The ZQL solution requires 20 megabytes for 2,500 clients, and 8 megabytes for 1,000

clients, making it untenable given that the clients need to transmit proofs frequently over cellular networks.

Figure 9(i) shows the proof size for the NIDS application as the intrusion trace length increases. The Pinocchio proof is about 1 KB, but again the tradeoff in latency makes this characteristic mostly irrelevant. The sizes for the ZØ and ZQL solutions are both linear, with the ZØ solution offering a savings of about 4× at all data points. This is a significant savings, considering that false positives may be frequent, so the client may need to send proofs to the server almost continuously throughout service.

8 Related Work

Tier-Splitting and Language Methods: A number of compilers exist that enable automated tier-splitting in some form. In the context of web programming, Google Web Toolkit (GWT) [21], Volta [26], Links [12], and Hilda [45] are among the pioneering efforts. ZØ is closest to Volta and GWT, allowing developers to supply a single piece of code that is compiled into separate modules for the client and server. Unlike those projects, ZØ uses cost models of execution time and data size to derive an optimization problem whose solution represents an ideal division of functionality between tiers.

Others have used tier splitting to provide security and privacy guarantees. SWIFT [11] builds on the JIF [30] language, incorporating security types for confidentiality and tier-splitting for web applications. To accomplish this, information flow constraints are embodied in an integer programming problem whose solution corresponds to a valid (e.g., secure) placement of code onto tiers that minimizes the number of messages that must be transferred. Unlike ZØ, SWIFT does not explicitly account for data size and transfer time when looking for a split that is likely to maximize performance.

Backes *et al.* [3] presented a compiler for distributed authorization policies written in Evidential DKAL [7], an authorization logic that supports signature-based proofs. The use of zero-knowledge proofs allows principals to prove access rights based on sensitive data without directly revealing its content. ZØ differs in its applicability: ZØ allows developers to use C# as part of a larger .NET application, whereas this work translates authorization logic formulas into cryptographic code.

Others have addressed the problem of untrusted client-side computation in various contexts [22, 23, 42, 44]. A similar notion of integrity was presented in Ripley [42], which prevents client-side cheating in web applications by efficiently replicating client-side computations on the server. Unlike ZØ, Ripley's mechanism does not preserve privacy.

Zero-Knowledge Proofs: Zero-Knowledge proofs of knowledge [6] have been extensively studied. Schemes have been developed for various types of relations and computations [8, 9, 20, 38]. Several projects have sought to provide zero-knowledge compilers [2, 3, 17, 28, 33] that take a proof goal and produce executable zero-knowledge code. The first set of zero-knowledge compilers [2, 3, 28] required specifications of cryptographic protocols [10], and so are difficult for non-cryptographers to use. The second generation [17, 33] are geared towards generating ZK code for general computations expressed in restricted high-level languages. Our work makes extensive use of these compilers to optimize zero-knowledge computation. There are a number of larger projects that incorporate zero-knowledge proofs in order to manage integrity without sacrificing privacy. Applications include privacy-preserving smart metering [37], random forest and hidden Markov model classification [13], and privacy-preserving automotive toll charges [4].

9 Conclusions

This paper paves the way for using zero-knowledge techniques for day-to-day programming. We have described the design and implementation of ZØ, a distributing zero-knowledge compiler which produces distributed applications that rely on ZKPK to provide simultaneous guarantees for privacy and integrity. We build on recent developments in zero-knowledge cryptographic techniques, exposing to the developer the ability to take advantage of these advances without requiring domain-specific knowledge or learning a new specialized language. Most of the heavy lifting is done by the compiler, including cost modeling to decide which zero-knowledge back-end to use and how to split the application for optimal performance, together with the actual code splitting.

Our cost-fitting models provide an excellent match with the observed performance, with R^2 scores at least and .98. Our global application optimizer is fast, completing in under 3 seconds on all programs. Our manual and experimental examination of program splits and back-end choices proposed by ZØ confirms that they are indeed optimal. Using six applications based on real-life commercial products, we show how ZØ makes it viable to use zero-knowledge technology. We observe performance improvements of over 40×. Perhaps most importantly, ZØ allowed many of the applications to scale to large data sizes with thousands of users while remaining practical in terms of computation time and data size. This means that applications which were not feasible using state-of-the-art zero-knowledge tools are now practical in realistic settings.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In *Proceedings of the European Conference on Research in Computer Security*, 2010.

[3] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[4] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Pretp: privacy-preserving electronic toll pricing. In *Proceedings of the Usenix Security Conference*, 2010.

[5] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Pretp: Privacy-preserving electronic toll pricing. In *Proceedings of the Usenix Security Symposium*, 2010.

[6] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *Proceedings of the International Cryptology Conference on Advances in Cryptology*, 1993.

[7] A. Blass, Y. Gurevich, M. Moskal, and I. Neeman. Evidential authorization*. In S. Nanz, editor, *The Future of Software Engineering*. 2011.

[8] S. Brands. Rapid demonstration of linear relations connected by boolean operators. In *Proceedings of the International Conference on Theory and Application of Cryptographic Techniques*, 1997.

[9] J. Camenisch, R. Chaabouni, and A. Shelat. Efficient protocols for set membership and range proofs. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, 2008.

[10] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *Proceedings of the International Cryptology Conference on Advances in Cryptology*, 1997.

[11] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure Web applications via automatic partitioning. *SIGOPS Operating Systems Review*, 41(6), 2007.

[12] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*. Springer Berlin / Heidelberg, 2007.

[13] G. Danezis, M. Kohlweiss, B. Livshits, and A. Rial. Private client-side profiling with random forests and hidden Markov models. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, 2012.

[14] D. Davidson, M. Fredrikson, and B. Livshits. MoRePriv: Mobile OS Support for Application Personalization and Privacy (Tech Report). Technical Report MSR-TR-2012-50, Microsoft Research, May 2012.

[15] C. Duhigg. How companies learn your secrets. `http://nyti.ms/SZryP4`, Feb. 2012.

[16] T. Fechner and C. Kray. Attacking location privacy: exploring human strategies. In *Proceedings of the Conference on Ubiquitous Computing*, 2012.

[17] C. Fournet, M. Kohlweiss, and G. Danezis. Zql: A compiler for privacy-preserving data processing. In *Usenix Security Symposium*, 2013.

[18] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. In *IEEE Symposium on Security and Privacy*, May 2011.

[19] F. D. Garcia, E. R. Verheul, and B. Jacobs. Cell-based roadpricing. In *Proceedings of the European Conference on Public Key Infrastructures, Services, and Applications*, 2012.

[20] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In *Proceedings of the IACR Eurocrypt Conference*, 2013.

[21] Google Web Toolkit. `http://code.google.com/webtoolkit`.

[22] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley Professional, 2007.

[23] S. Jha, S. Katzenbeisser, and H. Veith. Enforcing semantic integrity on untrusted clients in networked virtual environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[24] F. Kerschbaum. Privacy-preserving computation (position paper). `http://www.fkerschbaum.org/apf12.pdf`, 2012.

[25] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the International Conference on Mobile systems, Applications, and Services*, 2012.

[26] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Softtware*, 25(5):53–59, 2008.

[27] M. Marchetti, M. Messori, and M. Colajanni. Peer-to-peer architecture for collaborative intrusion and malware detection on a large scale. In *Proceedings of the International Conference on Information Security*, 2009.

[28] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: a language-based system for efficient zero-knowledge proofs and elec-

tronic cash. In *Proceedings of the Usenix Conference on Security*, 2010.

[29] Microsoft Research. Common compiler infrastructure. `http://ccimetadata.codeplex.com`, 2012.

[30] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[31] A. Narayanan and V. Shmatikov. Robust deanonymization of large sparse datasets. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[32] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[33] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[34] C. Pontoriero. Is groupon a raw deal for publishers? `http://risnews.edgl.com/retail-trends/Is-Groupon-a-Raw-Deal-for-Retailers-73442`, June 2011.

[35] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes, 3rd edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[36] J. Rattz and A. Freeman. *Pro LINQ: Language Integrated Query in C# 2010*. Apress, 2010.

[37] A. Rial and G. Danezis. Privacy-preserving smart metering. In *Proceedings of the Workshop on Privacy in the Electronic Society*, 2011.

[38] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[39] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2010.

[40] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In P. Ning and T. Yu, editors, *Proceedings of the 2007 ACM Workshop on Privacy in the Electronic Society, WPES 2007*, pages 99–107. ACM, 2007.

[41] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In *Proceedings of the ACM Workshop on Privacy in electronic society*, WPES '07, 2007.

[42] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing distributed Web applications through replicated execution. In *Conference on Computer and Communications Security*, Oct. 2009.

[43] Wikipedia. Usage-based insurance. `http://en.wikipedia.org/wiki/Usage-based_insurance`, 2013.

[44] J. Yan. Security design in online games. In *Proceedings of the Annual Computer Security Applications Conference*, 1993.

[45] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven Web applications. In *Proceedings of the International Conference on Data Engineering*, 2006.

Appendix

In the appendix, we present more details for each of our applications (Section A), show how a subset of C# is translated to each zero-knowledge backend (Section B), and give a further technical details on our global optimization algorithm (Section C).

A Motivating Case Studies

This section presents six case studies. Below we present a taxonomy of our applications along six dimensions that relate to practical deployment concerns.

Trusted Hardware Does this application require trusted hardware to establish integrity for the inputs? The three applications that do not have this requirement make a different trust assumption: the source of the data is trusted by the verifier, but not able to violate their privacy concerns.

Distributed Computation Does this application require some kind of peer-to-peer distributed computation? Both Slice and the Traffic Density application require multiple provers to share intermediate data using peer-to-peer communications.

Streaming Computation Does this application require the ZQL portion of the implementation to continuously accept and process input data, providing the verifier with a continuous stream of results and proofs?

Multiple ZQL Stages Does this application require rounds of iterated ZQL computations, interleaved with intermediate processing outside of ZQL? For example, both Slice and Traffic require an initial round of secret share generation (in ZQL), followed by a peer-to-peer transmission of shares (not in ZQL), followed by a round of share aggregation (in ZQL). This property suggests the need for a unified development and compilation framework, that takes care of the transitions between these stages.

Parallelizable Is this application inherently parallelizable? Two applications, Loyalty and CNIDS, are marked as "maybe" because their primary functionality relies on some form of automaton evaluation. This type of functionality may be parallelizable if an extended form of lookup table is eventually supported by the ZQL compiler.

New Primitives Does this application require new primitive support from the ZQL compiler? Three applications require either map2, fold2, or both.

Application	Description	Trusted hardware	Distributed computation	Streaming computation	Multiple ZQL stages	Parallelizable computation	New primitives
FitBit	A.1	✓	✗	✗	✗	✓	✗
Studies	A.2	✓	✗	✓	✗	✓	✓
Loyalty	A.3	✗	✗	✗	✗	?	✓
Waze	A.4	✓	✓	✓	✓	✗	✓
CNIDS	A.5	✓	✗	✓	✗	?	✗
Slice	A.6	✗	✓	✗	✓	✓	✓

Figure 12: Case studies: a classification and a guide.

Figure 12 shows a classification of our case studies along the dimensions outlined above. In each cell, ✓ corresponds to yes, ✗ to no, and ? to maybe.

A.1 Walk for Charity with FitBit

Several programs exist for paying users for the amount of exercise they perform, either directly in the form of rewards, or indirectly by making charitable donations on their behalf. This works by requiring users to log their exercise habits using a FitBit or other sensor device (e.g., a GPS-enabled tracker) to measure the distance the user walks, runs, or bikes, and send the logs to a centralized server.

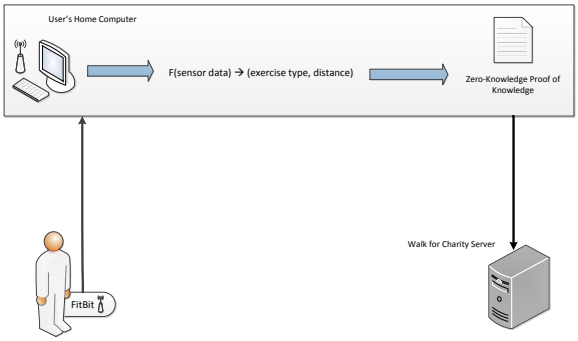
Privacy Concern: The user may not want to reveal their exercise route to a relatively unknown/untrusted third party.

Integrity Concern: The service is spending money on the basis of distance derived from sensor logs. If the logs can be tampered with, or the distance computation can be subverted, the possibility for fraud arises.

Proposed Solution: Keep all sensor readings local to the user's desktop machine, perform the distance computation locally on the desktop machine, send the result of the distance computation to the centralized third-party server. Use ZKPK to ensure that the distance computation is performed correctly. This approach is similar to what has been advocated for smart metering [37]. Note that we assume that the sensor readings are trusted, but the (desktop) machine that performs the distance computation is not.

The algorithm proceeds as follows:

1. After each GPS reading, the user's fitness device sends an encrypted commitment to the Walk for Charity server, as well as the user's home computer.



2. At the end of the day (or during some other downtime), ZQL code running on the user's computer processes the GPS readings, and computes the total distance walked by the user.

3. The results of the ZQL computation, as well as their correspond proof, are sent to the Walk for Charity server for validation.

A.2 Supervised Studies

Many scientific studies, especially in medical and social sciences, require subjects to wear sensors and undergo protocols that provide information about their physiological and psychological state. For example, a study that seeks to understand the effect of common workplace events on worker stress levels might require a participant to wear a galvanic skin response sensor, a camera to detect face-to-face interactions, and to complete regular surveys that measure psychological state.

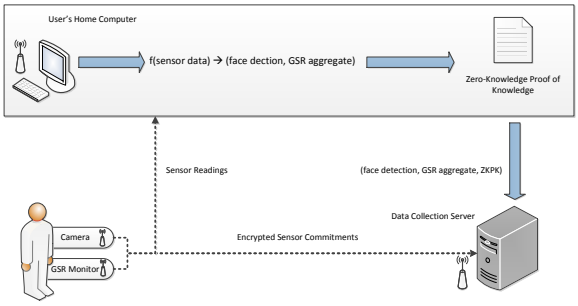
Privacy Concern: Participants may have concerns about the use of their physiological measurements, survey responses, or, most prominently, the processing of images taken from their cameras.

Integrity Concern: These studies typically involve payment given to subjects. Subjects concerned about their privacy, or those who simply do not want to wear intrusive sensor devices and take the time to complete surveys, have an incentive to *fake* the data used in the study.

Proposed Solution: Have all sensors associated with the study report readings to the subject's workstation. The workstation takes these readings and performs the aggregate computations relevant to the actual study, reporting the results and throwing the raw sensor readings out. ZKPK is used to ensure that the sensor readings are processed correctly. This assumes that the sensors attached to the subject are trusted, but that the subject's workstation is not.

The algorithm proceeds as follows:

1. At each time interval t, the sensors attached to the subject's body take a reading, and send it to the subject's workstation, as well as sending an encrypted commitment to the data collection server.



2. The subject's workstation performs aggregate computations over the readings, and sends the results, along with a zero-knowledge proof of correctness, to the data collection server.

   - The images are processed using a face detection algorithm based on Principal Components Analysis. A set of "eigenfaces" and their corresponding eigenvalues are assumed public input to the algorithm, and come pretrained from outside data. The algorithm simply projects the image from the camera into "face space", and computes its distance from the average face computed from the training set. The distance is returned from the computation.
   - At the moment, the GSR readings are fed through an identity map, as the primary threat to privacy in this scenario is currently presumed to be the images taken from the subject's camera. However, this aspect of the algorithm could be changed to return the mean or mode of some public number of samples.

3. The data collection server collects the proofs, and verifies them against the encrypted commitments sent by the sensors.

A.3 Personalized Loyalty Cards

Many of today's large retailers such as Target, BestBuy, etc. use customer loyalty cards to encourage repeat visits. Typically, the customer must enroll in a loyalty program, and receive a card that can be applied to receive discounts in future visits. Recently, certain retailers (e.g., Safeway) have begun personalizing this process by using the customer's past purchase history (available because of the association between checkout and loyalty card) to create discounts available only to one particular customer. Depending on the retailer, these discounts can be

sent to the customer's mobile phone, or applied automatically at checkout.

Privacy Concern: Many people are not comfortable with a retailer tracking their purchases. This is most readily illustrated by a recent scandal with Target discovering that a teenage girl was pregnant before her parents did [15].

Integrity Concern: Retailers offer discounts completely on the basis of past purchase history. If a customer were able to fake a purchase history, they might be able to obtain a discount for an item of their choosing. Moreover, having a reproducible strategy for "generating" discounts might create a serious problem for the retailed, similar to those experienced by some retailers that were overly generous in offering Groupon discounts [34].

Proposed Solution: The "loyalty app" on the customer's phone takes the place of the traditional card. At checkout, the app uses a near-field communication sensor with the register to receive a list of purchased items. This information is stored locally, and never sent to the store's servers. Personalization algorithms are applied to this sensor data on the phone, and the results are discounts that can be used at the next transaction. These discounts are transmitted to the register at the time of purchase, and ZKPK is used to demonstrate that the correct algorithms were applied to the NFC sensor data received in previous transactions. This assumes that the sensor readings are trusted, but the mobile app is not.

The algorithm proceeds as follows:

1. Before checkout time (possibly during the phone's downtime, or on the user's workstation), a ZQL query is executed to associate the user's previously certified transactions (see Step 2) with a set of discounts offered by the store. This is performed by processing the user's transaction history with a finite-state transducer. This assumes that the user gives his transaction history to the ZQL query in a particular order, which is checked by the query. The choice of using a finite-state transducer to associate discounts with transaction histories makes this model general with respect to the store's system of discounting. Decision trees and arbitrary sets of rules can also be encoded using this construct.

2. At checkout time, the user waves his smartphone at the register. The register certifies the GUID of each item purchased, and sends encrypted commitments to the smartphone via NFC, as well as recording them in a central database. The user's phone does not transmit any identifying information to the register, so the store is unable to associate the purchases with the user.

3. After receiving the certified transaction list for the current purchases, the user's smartphone sends the result of the transducer from Step 1, and its corresponding proof. The register can then provide the discounts.

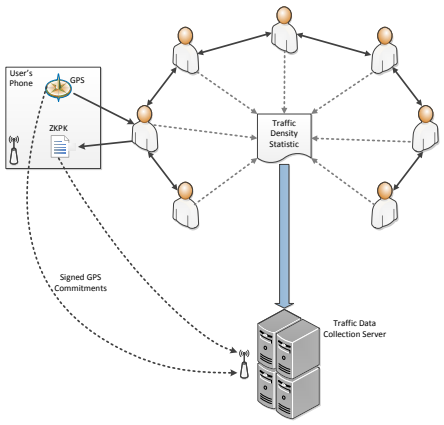
A.4 Crowd-sourced Traffic Statistics

There are several mobile applications that provide traffic congestion information to end-users based on the combined GPS readings of everyone using the app.

Privacy Concern: Users do not want to share their location with the app's servers, or the general public (in the case of a distributed protocol).

Integrity Concern: The app needs reliable GPS readings from users to provide its core functionality. If users wish to "game" the system by providing fake GPS readings (thus ensuring their privacy) while receiving the end-product, a tragedy of the commons scenario results.

Proposed Solution: Let the users keep their GPS readings local, and take part in a distributed protocol to compute local density information for transmission to the app's central server. ZKPK is used to ensure the integrity of the distributed protocol.



In more detail, this algorithm works as follows:

1. At regular intervals, the collection server sends a request to each client for traffic density statistics. Density statistics are represented by partitioning the map into regions, and counting the number of clients in each region.

2. On receiving a request, each client:

   - Takes a GPS reading, and has it signed using a trusted subsystem in the operating system. An encrypted commitment of this reading is forwarded to the collection server.

- The client computes its region, and executes a ZQL query to: (1) check that its computed region number is correct; (2) compute a set of linear secret shares of its region number. Assuming R regions, the region numbers used by the algorithm are:

$$R^0, R^1, \ldots, R^{(R-1)}$$

- The client sends its secret shares to all other clients, along with the proof that each share was computed correctly.
- On receiving the other clients' secret shares, the client adds them all together, and forwards the result, along with the proof that the shares were added correctly, to the collection server.

3. On receiving all shares, the collection server interpolates to learn the sum computed by the clients. The server than then compute the number of clients in each region by converting it into its base-R representation.

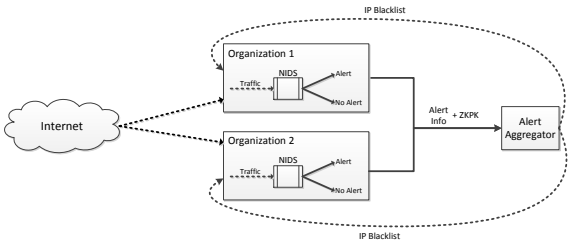
A.5 Collaborative Network Intrusion Detection

Collaborative intrusion detection (CNIDS) has been a longtime goal of security practitioners. In the CNIDS scenario, multiple (distrustful) organizations share the results of their network intrusion detection sensors, in order to provide their peers with advanced warning about possible threats. The most practical way to do this is to share IP blacklists — whenever an IP generates a valid NIDS alert on one organization's network, the IP is recorded and sent to the other participating organizations.

Privacy Concern: NIDS operate on highly sensitive data - raw network traces. Organizations participating CNIDS do not want to share their traces with other organizations, and in many cases, may be prohibited from doing so by law or organizational policy.

Integrity Concern: Given the privacy concern and the benefits of participating, some organizations may want to freeload by suppressing their own NIDS alerts. Additionally, if an adversary manages to compromise a participating network, it may choose to suppress or even generate false alerts, which may result in a denial of service for the targeted IP address.

Proposed Solution: Provide a ZKPK for the NIDS signature matching process, to show that network data is being correctly processed and reported. This will need to be done periodically (1) to demonstrate that alerts are not being suppressed and (2) because of the streaming nature of the application, and the consequent blow-up in proof size. Note that this approach assumes that raw network data coming into the NIDS is trusted, but that the machine performing the signature matching may not be.



The algorithm works as follows:

1. As network events come in, the NIDS machine certifies them, and sends encrypted commitments to a separate machine; for our purposes, call this machine the prover.

2. After a pre-defined number of network events have transpired, or an alert has been raised, the prover runs a ZQL query to traverse a finite state machine representing the NIDS signature using the certified network events. The query returns the IP address of the network event that caused the alert (if an alert resulted), or 0 otherwise. The prover sends the result of the traversal (the , as well as its ZK proof and the encrypted network traffic commitments, to the alert aggregator.

A.6 Slice: Organizing Shopping

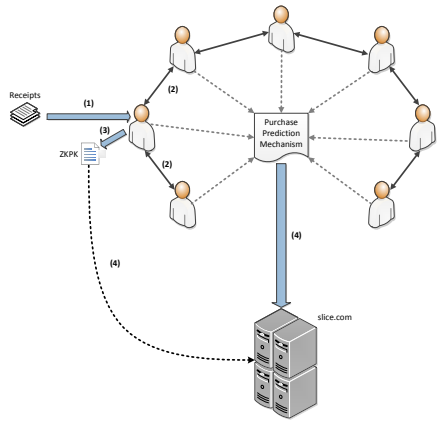
Slice is a service that takes as input a user's past purchase history, and provides various services using that data. One such service is product recommendation — given everybody's past purchase history, slice can build classifiers that predict a likely "next" purchase.

Privacy Concern: Handing one's entire purchase history to a profit-driven third party has obvious privacy implications.

Integrity Concern: Slice is providing a service to users who are willing to share their data, on the assumption that the data will be used to create a classifier that is valuable to both the user and Slice. A user, particularly one concerned about privacy, might provide fake data to Slice in order to obtain the useful classifier, which amounts to a type of fraud for the purposes of Slice.

Proposed Solution: Keep the user's purchase history local, and have the users take part in a distributed protocol in order to produce the classifier for Slice. Use ZKPK to ensure that no user is able to subvert the distributed classifier computation. This approach assumes that the purchase history data used by the distributed learning algorithm is trusted. Although it is not immediately clear how this end might be accomplished, one solution might

be to have the retailer certify each purchase, and send a commitment along with the user's receipt. This would achieve a similar level of trust to the current Slice implementation.



The approach produces a random forest classifier from the collective purchase history of all Slice users, with the goal of predicting whether a particular user is likely to purchase a given item. Assuming that we have a single, centralized database of purchase histories stored using the schema given above, a random forest would be constructed by the following algorithm (inspired by the relevant Wikipedia article):

For each tree, perform the following randomized computation:

1. Let m be the number of input variables used to determine the decision at each node of the tree.
2. Choose a random subset of n rows of the database to be used for training the current tree.
3. For each node of the tree, randomly choose m variables on which to split. Calculate the best split based on the MATT.
4. Value these variables take in the training set.
5. Grow the tree to a specified depth, and do not prune it.

The classifier is built by combining all trees. Classification is performed by traversing each tree for the given sample, and taking the statistical mode of the label associated with each traversed leaf node.

Our setting is slightly different: rather than having a centralized dataset, each row is housed on a different user's device. The users do not wish to share their row with Slice, so this algorithm must be run in a distributed fashion by sending queries to each user corresponding to the rows selected in step 2 of the algorithm given above. On receiving a query, the user invokes ZQL functionality (given below) to compute the correct answer based on their purchase history, and sends the query result and its zero-knowledge proof of correctness back to Slice. This is given in the following algorithm:

Our setting is slightly different: rather than having a centralized dataset, each row is housed on a different user's device. The users do not wish to share their row with Slice, so this algorithm must be run in a distributed fashion by sending queries to each user corresponding to the rows selected in step 2 of the algorithm given above. On receiving a query, the user invokes ZQL functionality (given below) to compute the correct answer based on their purchase history, and sends the query result and its zero-knowledge proof of correctness back to Slice. This is given in the following algorithm. For each tree, perform the following randomized computation:

1. Let m be the number of input variables used to determine the decision at each node of the tree.
2. Choose a random subset of n users to train the current tree. Label them $u_1, \ldots, u_n$.
3. For each node of the tree, randomly choose m variables on which to split. Label them $v_1, \ldots, v_m$.
   - Send a query q pertaining to each of the m variables to $u_1, \ldots, u_n$. Construct q by:

     $$q = l_1 \leq c_1 \leq u_1 \wedge l_m \leq c_m \leq u_m \wedge \ldots$$

     where c_j correspond to the amount spent in category j, and item i is the class label that we are trying to deduce.
   - Each user constructs a set of linear secret shares of his query result, and sends them to the other $u_1, \ldots, u_n$ (excluding himself), along with the proof that the share is constructed appropriately from the user's inputs.
   - The users add their shares independently, and send the result (and its ZK proof) to Slice.
   - Slice interpolates on the shares returned by the users, to obtain the number of users that match the query q.
4. Based on the query results given by $u_1, \ldots, u_n$, calculate the best split, and construct the new node.
5. Grow the tree to a specified depth, and do not prune it.

The classifier is built by combining all trees. Classification is performed by traversing each tree for the given sample, and taking the statistical mode of the label associated with each traversed leaf node.

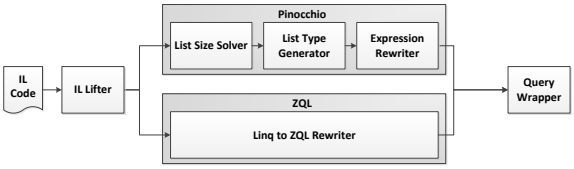


Figure 13: LINQ → translation process.

B Translating LINQ to Zero-Knowledge

In order to satisfy privacy and integrity constraints, our compiler translates some statements containing LinqExpr components in the worker methods into code that generates zero-knowledge proofs of knowledge. To accomplish this, ZØ relies on two *zero-knowledge back-ends*: ZQL [17] and Pinocchio [33]. Each back-end is itself a compiler, accepting as input an expression of a computation, and producing executable code to produce a zero-knowledge proof of the computation for a given set of inputs. As such, each back-end supports its own *expression language* with significantly different characteristics. The challenge addressed in this section is the translation of the common subset of LINQ supported by ZØ into the expression languages of these back-ends.

Figure 13 gives an overview of our back-end compilation process for ZQL and Pinocchio. The details differ significantly for each back-end, converging only on the first and last steps which correspond to lifting low-level intermediate language code into a higher representation and inserting I/O marshaling instructions before and after the compiled object code. This divergence of functionality is necessary given the differences between the two expression languages: ZQL's expression language is essentially a small subset of pure standard ML, whereas Pinocchio's is a subset of C with restrictions on data types and loop bounds. Because the subset of LINQ functions supported by ZØ corresponds to a small core of functional expressions, translating from ZØ to Pinocchio is much more involved than to ZQL.

B.1 Pinocchio

The structure of C code is substantially different from the types of Linq queries allowed by ZØ, and Pinocchio's additional restrictions make translation more complicated yet. First, all list sizes used in the Pinocchio expression must be statically-declared, and any operation over a list requires a static value to bound the corresponding loop statement. The LINQ commands in ZØ do not have these restrictions, so we must find a way to derive the needed information. Second, many expression forms in ZØ's LINQ commands have no corresponding expression form in C: they must be converted into statements whose side-effects are available as sub-expressions to enclosing expressions.

To perform translation to Pinocchio, ZØ follows a three-step process. First, static values for the size of each identifier that refers to a list value are derived using a constraint solver. The basis for this computation is a set of annotations provided by the developer, which indicate upper bounds on the sizes of certain input lists.

List Size Resolution: As previously discussed, Pinocchio requires static sizes for all lists and list operations, so our translation procedure requires a mapping from identifiers (for those that refer to list objects) to size constants. To produce such a mapping, we use a constraint resolution procedure over a set of bounding constraints generated by traversing the source expression. The rules for generating the constraints are given in Figure 14. Each rule is of the form $\Gamma, \textit{Syntactic Element} \Rightarrow \Gamma'$, where Γ and Γ' are sets of constraints. The constraints for each LINQ command are straightforward. The outcome of Select, Aggregate, and Zip operations has the same size as the input variable(s). The outcome of a First statement has the size of the elements contained in the input list.

The rules are invoked by a procedure that traverses each node of the program's AST, and performs syntactic matching on the entity represented by each node and the *Syntactic Element* of each rule. As the traversal proceeds, a list of constraints is maintained, and updated when rules match AST nodes. When the AST traversal completes, he set of constraints generated is passed to the Z3 SMT solver for resolution. If the constraints are satisfiable, Z3 will produce a *model*, which associates constraint variables to integers that satisfy the constraints. This model is used to derive the needed mapping between identifiers and list sizes.

Type Generation and Function Isolation: Pinocchio requires static sizes on all arrays and loop bounds. To accomplish this, ZØ creates a new `struct` type for each list with a distinct base type and size in the original program. Each new type has two fields: a static array and a constant defining the size.

Once types for each identifier are established, each sub-expression in the source statement is converted to a function body. To see the need for this step, consider the statement x.Select($el \rightarrow el$.Select(...)). C has no expression form for the functionality needed by the Select command, so both expressions must be converted into loop statements. Rather than placing the loop statements in the same method body and carefully managing side effects and sequencing with other sub-expressions, we isolate the emitted code for the inner Select in a separate function, and emit a call to the new function in its place in the context of the outer Select expression.

The statements generated for each LINQ command are straightforward translations of their defined behavior into basic C; in general, the input loop is iterated over, and the

$$
\begin{array}{ll}
con(expr) = & \\
\quad \{id.elt\} & \textbf{when } expr \text{ is } id.\mathsf{First}(\ldots) \\
\quad \{id_1, id_2\} & \textbf{when } expr \text{ is } id_1.\mathsf{Zip}(id_2, \ldots) \\
\quad \{id\} & \textbf{when } expr \text{ is } id.\mathsf{Aggregate}(\ldots) \\
\quad \{id\} & \textbf{when } expr \text{ is } id.\mathsf{Select}(\ldots) \\
\quad \{id.n\} & \textbf{when } expr \text{ is } id.\mathsf{Fld}(n) \\
con(id) = \{id\} &
\end{array}
$$

$$\text{C-FieldDef1}\ \frac{\varphi \leq id = \mathtt{x} \wedge id.elt = 1}{\Gamma, [\mathtt{MaximumInputSize(x)}]\ \mathsf{IEnumerable\langle T\rangle}\ id \Rightarrow \Gamma \cup \{\varphi\}}$$

$$\text{C-FieldDef2}\ \frac{\varphi = \begin{array}{l} id \leq \mathtt{x} \wedge id.elt \leq n_1 \wedge id.elt.elt \\ \leq n_2 \wedge \cdots \wedge id.(elt)^k \leq n_k \wedge id.elt^{k+1} = 1 \end{array}}{\Gamma, [\mathtt{MaximumInputSize(x, \{n_1, \ldots, n_k\})}]\ \mathsf{IEnumerable\langle T\rangle}\ id \Rightarrow \Gamma \cup \{\varphi\}}$$

$$\text{C-Method}\ \frac{id(id_1, \ldots, id_n)\ \text{ is a call site}}{\Gamma, \mathsf{Type}\ id(id_1^f, \ \ldots, \ id_n^f)\ \{\ldots\} \Rightarrow \Gamma \cup \{\mathtt{id}_1^f \geq \mathtt{id}_1, \ldots, \mathtt{id}_n^f \geq \mathtt{id}_n\}}$$

$$\text{C-New}\ \frac{V_i = con(expr_i)}{\Gamma, \mathtt{new}\ id(expr_1, \ldots, expr_n) \Rightarrow \Gamma \cup \bigcup_{1 \leq i \leq n} \{\bigwedge_{v \in V_i} id.i = v\}}$$

$$\text{C-Basic}\ \frac{Command \in \{\mathsf{Select}, \mathsf{First}\}}{\Gamma, id_1.Command(id_2 \rightarrow \cdots) \Rightarrow \Gamma \cup \{id_1.elt = id_2\}} \qquad \text{C-Aggregate}\ \frac{}{\Gamma, id_1.\mathsf{Aggregate}((id_2, id_3) \rightarrow \cdots) \Rightarrow \Gamma \cup \{id_1.elt = id_3\}}$$

$$\text{C-Zip}\ \frac{}{\Gamma, id_1.\mathsf{Zip}(id_2, (id_3, id_4) \rightarrow \cdots) \Rightarrow \Gamma \cup \{id_1.elt = id_3 \wedge id_2.elt = id_4\}} \qquad \text{C-Assign}\ \frac{V = con(expr)}{\Gamma, id = expr \Rightarrow \Gamma \cup \{\bigwedge_{v \in V} id = v\}}$$

Figure 14: List size constraint generation rules. Γ is a set of constraints.

lambda passed to the command is invoked over each element. Field lookups, new object construction, and function calls are rewritten to their C equivalents.

Pinocchio Example: Consider the example starting on line 33 of Figure 1. This command traverses the "discount" automata using a list of past purchases. In order to compile this to Pinocchio, we must first solve a set of constraints generated by a traversal of its syntax, as depicted in Figure 14. These constraints relevant to this command are given as:

1. $history \leq NPurchases$
2. $items.elt \leq NItems$
3. $automata \leq NEdges$
4. $transducer \leq NStates$
5. $purch\_state \leq history.elt.3$

Suppose that we instantiate

$$NPurchases = 500$$

$$NItems = 10,000$$

$$NEdges = 100$$

$$NStates = 50$$

The solver library solves these constraints in less than one second, and produces a model that allows us to resolve static sizes for all of the lists we need to perform the LINQ commands: Select, Aggregate, and First:

$$\{history = 500, items = 10000,$$
$$automata = 100, transducer = 50, purch\_state = 1\}$$

Using these sizes, ZØ generates the necessary input types to emit the LINQ commands. For the sake of brevity, we show only one such type used to represent the variable *automata*, as they all share a similar form:

```
struct Triple100 { Triple Enumerable[100]; }
```

ZØ then begins emitting new functions for each sub-expression in the source statement. For clarity, we will omit some function bodies by in-lining simple sub-expressions into certain function bodies; in reality, ZØ would emit a new function for *every* sub-expression. We begin with the lambda expression passed to the First command:

```
#define Boolean int
Boolean firstPredicate(Triple row, Int32 state,
    Int32 purch) {
  return row._1 == state && row._2 == purch;
}
```

This definition is used to construct the First command in a separate function: The definition `First` is used to generate the function for the Aggregate command that traverses the automata: The translation from LINQ to C for this command is straightforward: to aggregate a list, create

```
Pair First(Triple100 automata, Int32 state,
    Int32 purch) {
  int it;
  for(it = 0; it < Triple100Length; i++) {
    if(firstPredicate(
        automata.Enumerable[it], state, purch))
      return automata.Enumerable[it];
  }
  // Semantics is undefined when
  // Find cannot find the right element
  return automata.Enumerable[0];
}
```

```
Int32 Aggregate(Int32500 history,
    Triple100  automata) {
 int it, int0 = 0;
 for(it = 0; it < Pair10000Length; it++) {
  int0 = First(automata,
               int0,
               history.Enumerable[it]);
 }
 return int0;
}
```

an accumulator (`int0` in this case), and fold the aggregator function over each element in the accumulator in a loop that covers the entire list. Notice that in the actual code emitted by ZØ, this definition requires a separate function for each new object that is constructed; here, we in-line these functions into `Aggregate` to keep this discussion relatively brief. The entire query can be invoked by calling the resulting Pinocchio program with the relevant inputs to the original program, `history` and `automata`.

B.2 ZQL

Recall that we only attempt to convert LinqStmt statements into zero-knowledge, so there are four primary functions to convert, in addition to a few additional expression forms. By no coincidence, the four primary LINQ functions correspond closely to the operations supported by ZQL. Figure 15 gives a set of rewrite rules that can be used to translate a LinqExpr to ZQL's expression language. Select, Aggregate, Zip, and First calls are translated to map, fold, map2, and find expressions. Lambda definitions and functions calls are translated compositionally, by first translating sub-expressions and then building a new construct in the target language. Object creation using new is translated into tuple construction. Recall that user-defined types in a ZØ program must expose a single constructor that assigns all fields of the type; field names are translated into a tuple order using the constructor signature. Similarly, field accesses using fld are translated into a let binding that returns the appropriate tuple component; the translation consults the target identifier's type constructor to deduce the number of fields in the type.

Example: To illustrate the process of converting a ZØ LINQ statement to ZQL with rewrite rules, consider the example given in Figure 1. As previously discussed, the statement beginning on line 33 traverses an automata using the user's shopping history to arrive at a discount. Applying the rules from Figure 13, we start with T-Aggregate. The precondition of this rule states that both the initial accumulator and the lambda portions of our LINQ command must have valid ZQL translations. The initial accumulator is the constant 0, which is already valid ZQL.

Moving on to the lambda subexpression, we need to derive a translation for the expression body, which is another LINQ expression that performs a search using `First` over a list of triples. Descending recursively, we see that to translate the `First` command, we need to find a valid ZQL translation for the find predicate passed to the command. This is mostly straightforward, but requires an application of T-Fld to de-compose the `Triple` comprising each list element into its constituent `Int32` values. The only precondition of this translation is that the type of `Triple` has k fields for some k; this is true for $k = 3$. So, we can rewrite:

$$\texttt{trans.fld}\langle\texttt{int}\rangle((1)) \Rightarrow (\texttt{let}\ (\_1, \_2, \_3)\ =\ \texttt{trans}\ \texttt{in}\ \_1)$$

We can do the same for $\texttt{trans.fld}\langle\texttt{int}\rangle(2)$. The field accesses are used in a conjunctive equality test, which is translated compositionally using T-Op. With these rewrites, the final result for the find predicate is:

```
fun(trans) →
      (let(_1, _, _) = transin(_1 = state))
      @&(let(_, _2, _) = transin(_2 = purch))
```

Plugging this expression back into T-Aggregate, we arrive at the following for our final rewrite:

```
                                                    fold
(fun(state, purch) →
  find(fun(trans) →
  (let(_1, _, _) = transin(_1 = state))
  @&(let(_, _2, _) = transin(_2 = purch))
automata
```

This functionality is invoked on the input `history`; the expression is incorporated into an outer `query` function, which is called on LINQ to ZQL translations of each of the region's inputs.

C Global Optimization Details

The rules for performing global optimization are given in Figure 16. The top portion of Figure 16 specifies the inference rules needed to generate the privacy and functionality constraints, and the bottom portion the objective function used to characterize the suitability of a solution to the constraints. The rules are applied as part of a traversal of the program's abstract syntax tree. Each inference rule either updates the set of constraints collected for a program, or a *context* Γ; they are of the form:

$$\frac{\textit{Antecedent}}{\Gamma \vdash C, \textit{Pattern} \Rightarrow C'} \qquad \frac{\textit{Antecedent}}{\Gamma \vdash C, \textit{Pattern} \Rightarrow \Gamma'}$$

Γ is the *context* of the analysis, and tracks which method the traversal is currently in, as well as whether the traversal is in a zero-knowledge region and which external methods have an affinity to a particular tier. C is a set of constraints. *Pattern* is an AST pattern, such as $v = f(v_1, \ldots)$ to match an assignment. *Antecedent* is a pre-condition for using a rule: whenever *Antecedent* is true, then either the set of constraints C is updated to a new set C', or Γ is updated to a new context Γ', according to the specifics of the rule.

The rules in Figure 16 use a variable p_v for each program variable v to indicate the privacy level of v; levels correspond to integers that are mapped to tiers by the function ϕ. ϕ maps the tier *Any* (corresponding to the constraint that a variable may appear on any tier) to the value 0, and all other locations to positive integers. Similarly, a variable p_f is created to track the execution location of each worker method f. The constraints also create a variable c_{f_1,f_2} for each pair of worker methods

$$\text{T-Select}\ \frac{lambda \Rightarrow lambda_{\text{zql}}}{Id.\text{Select}(lambda) \Rightarrow (\text{map}\ (lambda_{\text{zql}})\ Id)} \qquad \text{T-Aggregate}\ \frac{lambda \Rightarrow lambda_{\text{zql}} \quad expr \Rightarrow expr_{\text{zql}}}{Id.\text{Aggregate}(expr,\ lambda) \Rightarrow (\text{fold}\ (lambda_{\text{zql}})\ expr_{\text{zql}}\ Id)}$$

$$\text{T-Zip}\ \frac{lambda \Rightarrow lambda_{\text{zql}}}{Id_1.\text{Zip}(Id_2,\ lambda) \Rightarrow (\text{map2}\ (lambda_{\text{zql}})\ Id_1\ Id_2)} \qquad \text{T-First}\ \frac{lambda \Rightarrow lambda_{\text{zql}}}{Id.\text{First}(lambda) \Rightarrow (\text{findt}\ (lambda_{\text{zql}})\ Id)}$$

$$\text{T-Lambda}\ \frac{expr \Rightarrow expr_{\text{zql}}}{(Id_1, Id_2, \dots)\ \rightarrow expr \Rightarrow \text{fun}\ (Id_1, Id_2, \dots)\ \rightarrow expr_{\text{zql}}} \qquad \text{T-Call}\ \frac{expr_i \Rightarrow expr^i_{\text{zql}}}{Id(expr_1, \dots, expr_n) \Rightarrow Id(expr^1_{\text{zql}}, \dots, expr^n_{\text{zql}})}$$

$$\text{T-Fld}\ \frac{\text{typeof}(Id)\ \text{has}\ k\ \text{fields}}{Id.\text{fld}(n) \Rightarrow (\text{let}(Id_1, \dots, Id_n, \dots, Id_k) = Id\ \text{in}\ Id_n)} \qquad \text{T-NewNamed}\ \frac{expr_i \Rightarrow expr^i_{\text{zql}}}{\text{new}\ Id(expr_1, \dots, expr_n) \Rightarrow (expr^1_{\text{zql}}, \dots, expr^n_{\text{zql}})}$$

$$\text{T-NewAnon}\ \frac{expr_i \Rightarrow expr^i_{\text{zql}}}{\text{new}\ \{Id_1 = expr_1, \dots, Id_1 = expr_n\} \Rightarrow (expr^1_{\text{zql}}, \dots, expr^n_{\text{zql}})}$$

Figure 15: Transformation from ZØ LINQ to ZQL expressions.

f_1 and f_2. c_{f_1,f_2} takes the value 0, except whenever there is a *tier crossing* between f_1 and f_2, meaning f_2 uses a value computed by f_1, and f_1 and f_2 do not reside on the same tier, in which case it takes the value 1. Finally, each program variable v is associated with two additional constraint variables z_v and q_v, corresponding to whether the zero-knowledge proof of v is computed by ZQL (in which case $z_v = 1$) or Pinocchio (in which case $q_v = 1$). z_v and q_v are mutually exclusive, i.e. $z_v = 1 \oplus q_v = 1$, as the proof of each variable is computed by at most one zero-knowledge engine. If v is not defined inside of a zero-knowledge region, then $z_v = q_v = 0$.

The rules propagate privacy concerns among the variables in a straightforward fashion: for an assignment $v = f(v_1, \dots, v_n)$ or $v = v_1\ op\ v_2\ op\ \dots\ op\ v_n$, the constraints are updated to reflect that either $p_v = p_{v_i}$ or $p_{v_i} = \phi(Any)$, for all v_i on the right-hand side of the assignment. Intuitively, either v has the same privacy requirements as v_i, or v_i does not have any privacy requirements at all. Similarly, whenever a variable v is referenced in a worker method f, either f must be placed at the tier matching the privacy requirement of v, or v must have no privacy requirement.

Whenever v is assigned in a zero-knowledge region, we constrain its privacy requirement to *Any*, effectively *declassifying* v. Whenever v is the target of an assignment whose right-hand side invokes external code, we assume that v must remain private to the tier on which its host worker method executes. This is a conservative over-approximation, based on the possibility that external code can perform arbitrary actions outside the purview of ZØ's analysis capabilities, such as leaking sensitive files or memory into return values.

The objective function in Figure 16 can be understood in two parts. The first part corresponds to the communication cost of any tier crossings in the tier partition: for each pair of worker methods f_i, f_j, the crossing variable c_{f_i,f_j} is multiplied by the cost of sending data between the tiers $\text{ComCost}(p_{f_i}, p_{f_j})$ and the size of the proofs that need to be communicated between the functions. The proof size is computed using cost models, as described in Section 4. Recall that c_{f_i,f_j} is zero except in solutions where f_i and f_j are placed on different tiers.

The second part of the objective function corresponds to the cost of building and verifying zero-knowledge proofs using the engines selected by the current solution. For each variable, there is a term corresponding to its proof generation and verification cost for each engine, multiplied by the cost of computation at the corresponding tier. As with the tier crossing variables c_{f_1,f_2}, z_v and p_v are zero except when the solution selects ZQL or Pinocchio for the variable v, respectively, so each term will only contribute to the final cost when the solution selects a particular zero-knowledge engine.

$$\frac{}{\Gamma \vdash C, [\texttt{Private}(Loc)]\ \texttt{Type v} \Rightarrow C \leftarrow C \cup p_v = \phi(Loc)}$$

$$\frac{}{\Gamma \vdash C, \texttt{Type MethodName}\ (...)\{...\} \Rightarrow \Gamma \leftarrow \Gamma[\mathsf{CurMethod} \mapsto \texttt{MethodName}]} \qquad \frac{\Gamma(\mathsf{ZKRegion}) = \mathsf{False}}{\Gamma \vdash C, v_1 = v_2 \Rightarrow C \leftarrow C \cup p_{v_1} = p_{v_2}}$$

$$\frac{}{\Gamma \vdash C, \texttt{ZKBegin}() \Rightarrow \Gamma \leftarrow \Gamma[\mathsf{ZKRegion} \mapsto \mathsf{True}]} \qquad \frac{}{\Gamma \vdash C, \texttt{ZKEnd}() \Rightarrow \Gamma \leftarrow \Gamma[\mathsf{ZKRegion} \mapsto \mathsf{False}]}$$

$$\frac{\Gamma(\mathsf{ZKRegion}) = \mathsf{False} \quad f \text{ is a worker method} \quad v_i \text{ is defined by return value of } f_i}{\Gamma \vdash C, v = f(v_1, \ldots, v_n) \Rightarrow C \leftarrow C \cup \left(\bigwedge_{1 \le i \le n} (p_{v_i} = p_f \vee p_{v_i} = \phi(Any)) \wedge (p_f \neq pf_i \implies c_{f_i,f} = 1)\right)}$$

$$\frac{\Gamma(\mathsf{ZKRegion}) = \mathsf{False} \quad f \text{ is an external method} \quad \Gamma(f.\texttt{ExecutionReq}) = l_{exec} \quad \Gamma(\mathsf{CurMethod}) = f}{\Gamma \vdash C, v = f(v_1, \ldots, v_n) \Rightarrow C \leftarrow C \cup p_v = p_f \wedge p_f = l_{exec}}$$

$$\frac{\Gamma(\mathsf{ZKRegion}) = \mathsf{False}}{\Gamma \vdash C, v = \texttt{new}\ f(v_1, \ldots, v_n) \Rightarrow C \leftarrow C \cup \bigwedge_{1 \le i \le n} p_v = p_{v_i} \vee p_{v_i} = \phi(Any)}$$

$$\frac{\Gamma(\mathsf{ZKRegion}) = \mathsf{True}}{\Gamma \vdash C, v = \star \Rightarrow C \leftarrow C \cup p_v = \phi(Any) \wedge z_v \in \{0,1\} \wedge q_v \in \{0,1\} \wedge z_v = 1 \oplus q_v = 1} \qquad \frac{\Gamma(\mathsf{ZKRegion}) = \mathsf{False}}{\Gamma \vdash C, v = \star \Rightarrow C \leftarrow C \cup \wedge z_v = 0 \wedge q_v = 0}$$

$$\frac{\Gamma(\mathsf{CurMethod}) = f}{\Gamma \vdash C, v \text{ is used} \Rightarrow C \leftarrow C \cup p_v = \phi(Any) \vee p_f = p_v}$$

$$\begin{aligned} \textit{Minimize} \quad & \sum_{f_i, f_j \in \text{Methods}} c_{f_i,f_j} \text{ComCost}(p_{f_i}, p_{f_j}) \text{DataSize}(f_i) \\ + \quad & \sum_{v \in \text{Variables}} z_v(\text{ZQLProver}(v)\text{TierComputeCost}(v, \textit{Prover}) + \text{ZQLVerify}(v)\text{TierComputeCost}(v, \textit{Verifier})) \\ + \quad & \sum_{v \in \text{Variables}} z_v(\text{PinocchioProver}(v)\text{TierComputeCost}(v, \textit{Prover}) + \text{PinocchioVerify}(v)\text{TierComputeCost}(v, \textit{Verifier})) \end{aligned}$$

Figure 16: Global optimization constraint generation and objective rules.