Modern distributed systems involve interactions between principals with limited trust, so cryptographic mechanisms are needed to protect confidentiality and integrity. At the same time, most developers lack the training to securely employ cryptography. We present Viaduct, a compiler that transforms high-level programs into secure, efficient distributed realizations. Viaduct’s source language allows developers to declaratively specify security policies by annotating their programs with information flow labels. The compiler uses these labels to synthesize distributed programs that use cryptography efficiently while still defending the source-level security policy. The Viaduct approach is general, and can be easily extended with new security mechanisms.

Our implementation of the Viaduct compiler comes with an extensible runtime system that includes plug-in support for multiparty computation, commitments, and zero-knowledge proofs. We have evaluated the system on a set of benchmarks, and the results indicate that our approach is feasible and can use cryptography in efficient, nontrivial ways.

CCS Concepts: • Security and privacy → Information flow control; Cryptography; Domain-specific security and privacy architectures.

1 Introduction

Modern distributed applications such as federated systems and decentralized blockchains typically involve parties from multiple administrative domains each with its own security policy. Companies might be required by law (such as the European Union’s GDPR [21]) to protect user privacy when they process user data or share it with other companies. The lack of full trust among parties makes it difficult to develop such systems, especially when the security requirements necessitate the use of cryptographic mechanisms. Recent efforts from the cryptography community have pushed these mechanisms from theory to practical deployment [5], but a gap remains: they still require too much expertise to use successfully [15, 17, 22].

We introduce Viaduct, a system that makes it easier for non-expert programmers to develop secure distributed programs that employ cryptography. It puts a variety of sophisticated cryptographic mechanisms in the hands of developers, including secure multiparty computation (MPC) protocols, zero-knowledge proofs (ZKP), and commitment schemes. Viaduct’s security-typed language allows developers to annotate programs with information-flow labels to specify fine-grained security policies regarding the confidentiality and integrity of data and computation. An inference algorithm allows these annotations to be lightweight, and enables Viaduct to reject inherently insecure programs. Viaduct then enforces these policies by compiling high-level source code to secure distributed programs, automatically choosing efficient use of cryptography without sacrificing security. The compiler supports a range of cryptographic protocols whose
security guarantees are characterized using information-flow labels. New protocols can be added to Viaduct by specifying their security properties and by implementing well-defined interfaces.

Although prior efforts have attempted to bridge this gap, most existing work focuses on compiling programs to a fixed set of cryptographic mechanisms. For example, some focus on compiling programs to MPC (e.g., Wysteria [41], Oblivio [34], SCALE-MAMBA [3]); others focus on ZKP (e.g., Pinocchio [38], Buffet [47], xjSNARK [32]). To our knowledge, by providing a unified abstraction to both specify security policies of programs and to specify security guarantees of cryptographic mechanisms, Viaduct is the first system to compile secure, distributed programs with an extensible suite of cryptography.

We make the following contributions:

- An algorithm to infer minimum consistent security requirements of data storage and computation for programs written in a security-typed language. (§3)
- A technique to compile secure distributed programs, deploying an extensible set of cryptographic protocols while minimizing a customizable notion of cost. (§4)
- An extensible runtime system for running compiled programs. Cryptographic mechanisms are added as plug-ins to the runtime. (§5, §6)
- An evaluation that shows that the Viaduct compiler can synthesize a wide variety of secure and efficient distributed programs, that the compilation technique is scalable, and that the annotation burden of the source language is minimal. (§7)
- An open-source implementation of the Viaduct compiler and runtime system.¹

2 Overview of Viaduct

Figure 1 gives a high-level overview of Viaduct. Its compiler takes a high-level source program partially annotated with information-flow labels. The compiler infers labels consistent with programmer-supplied annotations to determine security requirements for all program components. Then for each component the compiler selects a protocol that matches these requirements, guiding the selection with a cost model. The output is a secure and efficient distributed program, which hosts execute using the Viaduct runtime system. The Viaduct architecture has a small set of well-defined extension points, allowing developers to add support for new protocols with relative ease.

We give two examples to motivate and describe the Viaduct compilation process.

**Historical Millionaires’ Problem.** Our first example is a slightly modified version of the “millionaires’ problem” [48]. As in the classic formulation, two individuals, Alice and Bob, want to determine who has more money without revealing how much money they have to the other person. Rather than comparing their current wealth, in our “historical” variant Alice and Bob want to see who was richer at their poorest. Figure 2 shows an implementation of the historical millionaires’ problem in Viaduct. The program compares Alice’s lowest wealth with Bob’s, and outputs the answer (b richer) to both Alice and Bob.

![Figure 2.](https://github.com/apl-cornell/viaduct)

Figure 2. Implementation of the historical millionaires’ problem in Viaduct. Viaduct uses MPC for the comparison $a < b$, but computes the minima locally.

![Figure 3.](https://github.com/apl-cornell/viaduct)

Figure 3. Guessing game, where Alice attempts to guess Bob’s secret number. Viaduct uses zero-knowledge proofs so Alice learns nothing more than whether her guesses are correct. Most labels in this code can be inferred automatically.

![Available at](https://github.com/apl-cornell/viaduct)

1Available at https://github.com/apl-cornell/viaduct.
the confidentiality component of $B$). This label means that Alice fully trusts host $alice$ (with both confidentiality and integrity), while Bob trusts host $alice$ to execute the program correctly, but does not trust the host with his secret data.

All variables and expressions in Viaduct carry a security label, which is derived from the possible flows of information in the program. The variables in lines 4–7 carry the same label as their respective hosts, since they only involve data local to that host. However, the comparison $a < b$ involves both hosts’ private data, so has the higher security label $A \land B$. This label corresponds to data that is secret to and trusted by both principals. Since $A \land B$ corresponds to secret data, we require an explicit declassification to the label $A \land B$, which describes data that both hosts can see and trust.

During protocol selection (§4), Viaduct chooses cryptographic protocols to securely and efficiently execute our example. The central idea that allows Viaduct to select protocols automatically is that the security guarantees of protocols can also be captured by labels. Neither Alice nor Bob alone has enough authority to be responsible for the comparison, so Viaduct generates the following distributed implementation: Alice and Bob compute their respective minima locally but perform the comparison $a < b$ in semi-honest MPC. A semi-honest MPC protocol works here because the authority labels assigned to the hosts indicate that Alice and Bob trust each other’s hosts for integrity. Without that assumption, Viaduct is instead forced to select another protocol such as maliciously secure MPC.

There are typically multiple ways to assign protocols to a given program expression. For example, the computation of Alice’s minimum on line 6 could be securely performed in MPC, but since the computation requires the authority of Alice alone, it is cheaper yet still secure to do the computation locally on Alice’s machine. Using its cost estimator, Viaduct compiles the optimal program described above.

After protocol selection, Viaduct outputs a distributed program which captures the required cryptography to execute the source program. Hosts can execute this distributed program using Viaduct’s runtime system.

**Guessing Game.** Figure 3 presents a contrasting example. Here, Alice and Bob have security labels $A$ and $B$ respectively, modeling a malicious corruption scenario. Since they do not trust each other to execute the program correctly, semi-honest MPC is not applicable. Bob inputs a number $n$, and Alice has five attempts to guess the number. Since Bob’s input initially has label $B$, it must first be endorsed to the label $B \land A^\leftarrow$, raising integrity so that Bob cannot unilaterally modify the value. This endorsement requires a cryptographic mechanism to protect the integrity and secrecy of variable $n$ throughout program execution.

Viaduct synthesizes a program in which Bob commits to $n$ so that its value remains secret to Alice but Bob cannot later lie about the committed value. The statement $n == tguess$ is computed by having Bob send a zero-knowledge proof (ZKP) to Alice, so that Alice can trust the outcome but learns no additional information. All other variables are replicated in plaintext across the two hosts.

These examples show that Viaduct is general, as it treats protocols such as MPC and ZKP uniformly.

### 2.1 Specifying Security Policies

In Viaduct, security policies capture a notion of authority. Policies are represented by principals, formulas composed of conjunctions and disjunctions over a set of base principals $\{A, B, C, \ldots\}$ and two special principals $0$ and $1$. Principal $0$ represents maximal authority and corresponds to the conjunction of all base principals; principal $1$ represents minimal authority and corresponds to the disjunction of all base principals. We distinguish authority over confidentiality and over integrity. The security requirements of information are thus characterized by labels consisting of pairs $(p_c, p_i)$ of two principals $p_c$ and $p_i$, for confidentiality and integrity respectively.

A conjunction of principals $p_1 \land p_2$ represents combined authority. For confidentiality, this means the principal is allowed to read data that $p_1$ may read and also data that $p_2$ may read. For integrity, the conjunction may influence data that $p_1$ may influence, and also data $p_2$ may influence. A disjunction $p_1 \lor p_2$ corresponds to common authority, which may read or influence exactly the data that either $p_1$ and $p_2$ may individually.

Principals carry a natural partial order based on their authority. We write $p_1 \Rightarrow p_2$ to mean $p_1$ “acts for”, or is at least as trusted as, $p_2$. This relation coincides with logical implication: for example, $p_1 \land p_2 \Rightarrow p_1$ and $p_1 \Rightarrow p_1 \lor p_2$.

It is convenient to have syntax that works over both components of labels simultaneously. So, we extend $0, 1, \land, \lor$, and $\Rightarrow$ pointwise, and write one principal to mean that the two components are the same. For example, the annotation $(A)$ denotes the label $(A, A)$. To talk about confidentiality and integrity separately, we use projections, writing $\ell^\leftarrow$ for the confidentiality projection of $\ell$ and $\ell^\rightarrow$ for its integrity. Thus, $(B \land \land^\rightarrow)$ expands to $(B, B \land A)$, meaning Bob’s sole confidentiality and the combined integrity of Alice and Bob.

These projections are defined formally as follows:

$$\langle p_c, p_i \rangle^\leftarrow \equiv \langle p_c, 1 \rangle$$
$$\langle p_c, p_i \rangle^\rightarrow \equiv \langle 1, p_i \rangle.$$

The reflection operator [49] swaps the two components:

$$\mathbf{X}(\langle p_c, p_i \rangle) \equiv \langle p_i, p_c \rangle.$$

Viaduct programs assign labels to hosts to indicate the amount of trust placed in them, but there are also labels on data. The important insight, borrowed from FLAM [4], is that the same set of labels can be used to talk about both authority and information flow. When placed on data, a label takes on an information flow interpretation, specifying the minimum authority required to read and influence that data. As in
FLAM, standard operations from information flow literature can be reformulated in terms of authority:

\[ \ell_1 \subseteq \ell_2 \iff \ell_2^\rightarrow \Rightarrow \ell_1^\rightarrow \quad \text{and} \quad \ell_1^\rightarrow \Rightarrow \ell_2^\rightarrow \]  
(flows to)

\[ \ell_1 \sqcup \ell_2 \equiv (\ell_1 \land \ell_2)^\neg \land (\ell_1 \lor \ell_2)^\neg \]  
(join)

\[ \ell_1 \sqcap \ell_2 \equiv (\ell_1 \lor \ell_2)^\neg \land (\ell_1 \land \ell_2)^\neg \]  
(meet)

The flows-to relation \( \ell_1 \subseteq \ell_2 \) orders information flow policies: it means label \( \ell_1 \) is more permissive about the use of information than \( \ell_2 \). The join \( \ell_1 \sqcup \ell_2 \) is more restrictive than both \( \ell_1 \) and \( \ell_2 \), and the meet \( \ell_1 \sqcap \ell_2 \) is more permissive than either \( \ell_1 \) or \( \ell_2 \). The most restrictive label—that of completely secret, untrusted data—is \( 0^\neg = (0, 1) \), and the least restrictive (public, trusted data) is \( 0^\neg = (1, 0) \).

### 2.2 Threat Model

Compiled programs run in a distributed setting in which each host executes a single thread concurrently with other hosts. Hosts communicate via message passing over secure, private, asynchronous channels. There is no shared memory that spans multiple hosts. We assume the attacker cannot observe wall-clock timing. Additionally, we are not concerned with availability, so the attacker can halt execution at any time.

In the setting of Viaduct, there is no single notion of an attacker. For example, in the historical millionaires problem, neither Alice nor Bob fully trust the other. To Alice, Bob is a potential attacker; Alice expects her security requirements to be met as long as the behavior of Bob’s (partially trusted) host is accurately described by the label assigned to it \((B \land A^\neg)\). Conversely, to Bob, Alice is a potential attacker. Hence, we are concerned with security versus all possible attackers.

We model the power of an attacker using a label. The attacker can read the data on a host if the confidentiality of the attacker label is at least as trusted as that of the host, and can change data and code on the host if the integrity of the attacker label is at least as trusted as that of the host. We do not consider unreasonable attack scenarios in which a host has compromised integrity but still enforces confidentiality.\(^2\)

For example, in the historical millionaires’ problem, there are five interesting corruption scenarios: no corrupted hosts; Alice has corrupted confidentiality; Bob has corrupted confidentiality; both have corrupted confidentiality; or both Alice and Bob are fully corrupted. The full corruption of a single host is not possible because the hosts trust each other, so if the integrity of one is corrupted then the other’s integrity must be corrupted also.

### 2.3 Label Inference

Viaduct selects a protocol for every piece of data and computation in the program based on their authority requirements, represented as labels. Intuitively, program components must be executed by protocols with enough authority to defend the confidentiality of host inputs and the integrity of host outputs. These authority requirements are captured formally by a type system (§3.1), and Viaduct uses a novel inference algorithm (§3.2) to compute for all program components the minimum-authority labels that still respect the information-flow constraints on the program.

The only required label annotations on Viaduct programs are the authority labels on host declarations and labels on declassify/endorse expressions—all labels on variables can be elided, making annotation burden low. As we show in our evaluation, in practice these required annotations are enough to capture programmer intent: minimally annotated programs compile to the same distributed programs as their fully annotated versions.

### 2.4 Protocol Selection

After label inference, Viaduct performs protocol selection, which assigns a protocol to compute and store each subexpression and variable. Protocols encompass storage and computation performed “in the clear” as well as cryptographic mechanisms such as commitments, MPC and zero-knowledge proofs.

Each protocol \( P \) carries an associated authority label \( L(P) \), which approximates the security guarantees the protocol provides. Given a program component with minimum authority requirement \( t \), protocol selection only assigns \( P \) to execute that component if \( L(P) \Rightarrow t \)—that is, if \( P \) meets the authority requirement for the program component.

Intuitively, given a program \( s \) and protocol \( P \), we may imagine an ideal functionality \( P^s \) (in the style of UC \([9]\)) which executes the program fragments of \( s \) that are assigned to \( P \). The fragments of \( s \) that are assigned to \( P \) may depend on the computational abilities of \( P \). For example, if \( P \) is a commitment protocol, then \( P^s \) is only able to store values but not perform any computations. If \( P \) is an MPC protocol, then \( P^s \) can execute computations that can be translated into circuits—the standard interface for MPC implementations.

\( P^s \) guarantees that the storage and computation it performs are protected at label \( L(P) \). In particular, the adversary cannot observe storage or computation performed by \( P^s \) unless its confidentiality is at least \( L(P) \); dually, the adversary cannot influence storage or computation performed by \( P^s \) unless its integrity is at least \( L(P) \).

Examples of protocols and their corresponding authority labels are given in Figure 4. Following the above intuition for the security of functionalities \( P^s \), the authority label of protocols are determined to be the least authority required of the adversary to corrupt the protocol (in confidentiality or integrity). We explain the example protocols below:

Local\((h)\). No cryptography is performed, and data is stored and computations performed on host \( h \) in the clear. It provides exactly the authority of \( h \).

---

\(^2\) The semi-honest and malicious threat models common in cryptography correspond to corrupting only hosts’ confidentiality and corrupting both hosts’ confidentiality and integrity respectively.
### Figure 4. Example protocols and security labels that represent their authority.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Authority label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local((H))</td>
<td>(\mathcal{L}(h))</td>
</tr>
<tr>
<td>Replicated((H))</td>
<td>(\bigcap_{h \in H} \mathcal{L}(h))</td>
</tr>
<tr>
<td>Commitment((h_p, h_v))</td>
<td>(\mathcal{L}(h_p) \land \mathcal{L}(h_v)^-)</td>
</tr>
<tr>
<td>ZKP((h_p, h_v))</td>
<td>(\mathcal{L}(h_p) \land \mathcal{L}(h_v)^-)</td>
</tr>
<tr>
<td>MAL-MPC((H))</td>
<td>(\bigwedge_{h \in H} \mathcal{L}(h))</td>
</tr>
<tr>
<td>SH-MPC((H))</td>
<td>(I = \bigvee_{h \in H} \mathcal{L}(h)^-) \land I</td>
</tr>
</tbody>
</table>

MPC computation may be compromised if any host behaves maliciously. The confidentiality is equal to

\[
\left(\bigvee_{h \in H} \mathcal{L}(h)^-\right) \lor \left(\bigwedge_{h \in H} \mathcal{L}(h)^-\right).
\]

The first disjunct captures the fact that confidentiality guarantees are discarded if the integrity of any host is compromised. The second disjunct states that, if all hosts follow the protocol correctly, the adversary can only learn the state of intermediate MPC computations if all hosts have corrupted confidentiality. Overall, this means that in order to compromise confidentiality guarantees of semi-honest MPC, either the integrity of any host or the confidentiality of all hosts must be compromised.

In particular, for the historical millionaires’ example, the label of SH-MPC\((alice, bob)\) is \(A \land B\). This is because hosts alice and bob are both assumed to have the high integrity of \((A \land B)^-\). If alice and bob only have their own integrity, however, then the label is computed to be \(A \lor B\). The protocol only has enough authority to perform computations over data public to both hosts, and neither host trusts the result. Indeed, semi-honest MPC offers little to no benefit if any host has lower integrity than any other.

### 2.5 Runtime

Viaduct provides a modular runtime system for executing compiled distributed programs, implemented as an interpreter. All hosts run the interpreter with the same compiled program, which then executes each host’s portion of the program. During execution, the interpreter calls out to back ends implementing the cryptographic mechanisms used in the program. Back ends translate computations in the source language into their cryptographic realizations. For instance, the back ends for MPC and ZKP in our implementation build a circuit representation of the program as it executes.

Protocol back ends can send data to and receive data from each other, supporting the composition of protocols. Source-level declassification and endorsement induce this communication. For example, in Figure 2 on line 8, the computation \(a < b\) is declassified from label \(A \land B\) to \(A \lor B\). This declassification causes the MPC protocol between Alice and Bob to execute its stored circuit for this comparison, and to output the result in cleartext.

Figure 5 shows the execution of the program compiled by Viaduct for the historical millionaires’ problem. The program runs as follows. (1) First, the cleartext back ends on Alice and Bob’s machines receive input locally and compute their respective minima. The back ends send the minima as secret inputs to their respective MPC back ends, which create input gates for these inputs. (2) Next, the MPC back ends on Alice and Bob’s machines each create an operation gate that compares Alice and Bob’s secret inputs. The back ends jointly execute the circuit with the comparison result.
The syntax for Viaduct’s source language, a simplified version of the surface language, is given in Figure 6. The language supports base types such as booleans and integers, along with their usual operators. Surface-level assignables (val and var declarations) and arrays are uniformly represented as data types, which are restricted forms of objects. Like regular objects, they are created using constructors (new declarations) and contain methods. For simplicity, we only include three data types: immutable/mutable cells, which model surface-level assignables, and arrays. Arrays are dynamically sized but statically allocated: the size of an array can depend on values known only at run time, but array references cannot be rebound to different names or stored in arrays.

We distinguish between fully evaluated atomic expressions \( a \), and expressions \( e \) that evaluate to values and may have side effects. Methods include get and set operations for both mutable cells and arrays (for which they take an index as an extra argument). Input/output expressions allow programs to interact with hosts. The declassify expression marks locations where private data is explicitly allowed to flow to public data, while the endorse expression marks locations where untrusted data is explicitly allowed to influence trusted data.

Statements consist of let-bindings, assignable declarations, as well as the usual conditionals, loops, and sequential composition. Temporaries bind values while assignables bind instances of data types. We require all intermediate computations to be let-bound by a temporary, enforcing a variant of A-normal form [18]. We use the more general loop-until-break statements instead of the more traditional while loops, simplifying the conversion to A-normal form. A break statement (break \( b \)) includes an identifier \( b \) that names the loop it breaks out of. While loops are recovered easily:

\[
\text{while } e \text{ do } s \triangleq b : \text{loop } (\text{if } e \text{ then } s \text{ else break } b) .
\]

### 3.1 Label Checking

Viaduct’s type system enforces secure information flow in a standard way. The type system serves two purposes. First, it helps programmers ensure there are no unintended information flows: secrets are not leaked to and data is not corrupted by unauthorized principals. Second, it specifies what labels can be assigned to variables and expressions that the user did not explicitly annotate.

Figure 7 presents label checking rules for expressions and selected statements. Expressions are checked by the judgment \( \Gamma ; pc \vdash e : \ell \), which means that \( e \) has label \( \ell \) under the context on the left. Here, \( \Gamma \) is a finite partial map from temporaries, assignables, or loop names to labels:

Label Contexts \( \Gamma ::= \cdot | \Gamma, t : \ell | \Gamma, x : \ell | \Gamma, b : \ell \)

The program counter label \( pc \) is a standard way to prevent implicit flows of information via control flow [44]. The rules...
Viaduct

\[
\begin{align*}
\Gamma; a : \ell & \quad \Gamma; pc \vdash e : \ell \\
\Gamma; a : \ell & \quad \Gamma \vdash e : \ell \\
\Gamma; t : \ell & \quad \Gamma \vdash t : \ell \\
\Gamma; pc \vdash \text{declassify } a \text{ to } \ell & \quad \Gamma; pc \vdash \text{endorse } a \text{ to } \ell \\
\Gamma; pc \vdash \text{input}_p h : \ell & \quad \Gamma; pc \vdash \text{output } a \text{ to } h : \ell \\
\Gamma; pc \vdash s & \quad (\Gamma, t : \ell); pc \vdash s \\
\Gamma; pc \vdash \text{let } t \equiv e \text{ in } s & \quad \Gamma; pc \vdash a_1 : \ell \\
\Gamma; x : \ell; pc \vdash s & \quad \Gamma; pc \vdash \text{new } x \equiv D(a_1, \ldots, a_n) \text{ in } s \\
\Gamma; pc \vdash \text{if } a \text{ then } s_1 \text{ else } s_2 & \quad \Gamma; pc \vdash b : \ell \\
\Gamma; pc; b : \ell & \quad \Gamma; pc; \text{break } b \\
\Gamma; pc; s_1 & \quad \Gamma; pc; s_2 & \quad \Gamma; pc; \text{skip}
\end{align*}
\]

Figure 7. Information flow checking rules for expressions and statements.

for method calls and input/output expressions differ from those in standard security-typed languages in that they also include premises with \( pc \) checks. These checks are required because these expressions may induce communication between hosts, and hosts may learn secrets based on which requests they receive. Prior work that targets the distributed setting contains similar checks to control read channels [52]. Statement checking rules have the form \( \Gamma; pc \vdash s \); they are largely standard [44]. Because we assume attackers cannot observe timing nor analyze traffic, the rule for conditional statements does not require branches to have the same timing behavior or effects (e.g., method calls, input/output).

Nonmalleable Information Flow Control. Information flow type systems typically aim to enforce a compositional security property such as noninterference [23]. Noninterference is a strong property but it is too restrictive for practical applications, which usually have a more nuanced policy for secure information flow. Hence, like most languages supporting information flow control (e.g., [6, 37, 40]), Viaduct allows programmers to signify the exceptions to a noninterference policy through downgrading expressions.

Downgrading enables information flows that would violate noninterference, so it can be dangerous. This is especially true in the distributed setting, where storage and computation can be performed by hosts that one does not fully trust. Downgrading confidentiality (declassification) allows secret information to be treated as public information—a necessity for many applications, but doing so might allow a corrupted host to control when information is released or what information is released. Downgrading integrity (endorsement) allows untrusted information to be treated as trusted information, but might enable a corrupted host to trick an honest one into accepting mauled secrets.

The property of nonmalleable information flow control (NMIFC) [11] prevents both of these abuses of downgrading by combining two properties: robust declassification [51] and transparent endorsement [11]. Robust declassification requires that principals to which data is declassified could not have influenced either the decision to declassify or the data itself. Meanwhile, transparent endorsement prevents trusting mauled secrets by ensuring that information can only be endorsed if the providing principal can read it.

The declassification and endorsement rules in Figure 7 enforce NMIFC using the reflection operator \( \mathcal{X} \) (§2.1). The rules prevent the program from downgrading information with compromised labels [49], in which confidentiality exceeds integrity. These rules generate authority requirements that prevent the Viaduct compiler from placing data and computation on insufficiently trustworthy hosts. For example, consider a program where a server releases secret information to a client when the client guesses the correct password:

```plaintext
host server: (S), client: (1) val info: int(S), pw: int(S), guess: int(1) if (declassify (pw == guess) to (1)) output (declassify info to (1)) to client
```

This program violates robust declassification, because the decision to declassify info depends on (low-integrity) guess. Without the restrictions on downgrading, Viaduct could compile the program to store the guard \( pw == guess \) (with label 1) on the client. The client could simply claim to the server that its guess is correct! For this program to type-check with
We present an algorithm to infer these labels, which appear in rule premises, to acts-for ($\ell_t \leadsto \ell_s$) variables are initialized to $1_L$. Variables at acts-for ($\ell_L$) and unsatisfied constraints are used to update variables repeatedly, until a fixed point is reached, according to the rules in Figure 9. Constraints of the form $L_1 \Rightarrow L_2$ or $L_1 \Rightarrow L_2 \lor L_3$ are used to perform the corresponding update.

However, the rules in Figure 8 can also generate constraints of the form $L_1 \land p_2 \Rightarrow L_3$, arising from the typing rule for robust declassification. The term $p_2$ is always a constant since Viaduct requires annotations on declassify operations, so the value of $L_1$ can be updated safely to $L_3$, which denotes the weakest authority $p$ such that $p \land p_2 \Rightarrow L_3$. When a lattice supports the $\rightarrow$ operation, it is a Heyting algebra [43], allowing each update rule to lower the left-hand-side variable to the minimum authority satisfying the constraint. Any free distributive lattice, such as our lattice of principals, is a Heyting algebra. We prove this fact, as well as the fact that iterative analysis always terminates with the minimum-authority solution, in the supplemental technical report [2].

### 4 Protocol Selection

The protocol selection phase of Viaduct assigns a protocol to each program component. Formally, a protocol assignment is a function $\Pi : (T \cup X) \rightarrow P$ from temporaries and assignables to protocols. For a temporary $t$, $\Pi(t)$ is the protocol that executes the expression associated with $t$. Similarly, $\Pi(x)$ is the protocol that stores and responds to method calls on the data type instance bound to $x$.

#### 4.1 Validity of Protocol Assignments

Figure 10 outlines the conditions under which a protocol assignment is valid. The judgement $\Pi \models e : P$ means that expression $e$ can be executed by protocol $P$ under assignment $\Pi$. Similarly, the judgement $\Pi \models s$ means that $\Pi$ is a valid assignment for statement $s$.

We now describe the rules for validity. The rule for temporaries states that $t$ can only be read by protocol $P$ if $\Pi(t)$, the protocol storing $t$, can communicate with $P$, written $\text{comm}(\Pi(t), P)$. Not all pairs of protocols can communicate; the customizable protocol composer, discussed further in §5.1, defines the valid set of protocol compositions.

Other rules restrict where certain expressions can be executed. A method call on $x$ must be executed by $\Pi(x)$, the protocol that stores $x$. Similarly, input/output expressions must be executed locally on the relevant host.

The rules for temporary and assignable declarations ensure that the protocol selected for a temporary or assignable has enough authority to securely store it. Formally, the label $L(\Pi(t))$ of the protocol storing temporary $t$ must act for ($\Rightarrow$) the minimum required authority label $L(t)$ computed for $t$ in §3.2 (and similarly for assignables). Labels $L(\Pi(t))$ are the ones explained in Figure 4.

The rule for conditional statements ensures that all hosts involved in the execution of a conditional statement (Figure 11) can learn which branch is taken. The first premise

\[
\ell_t \leq \ell_s \leadsto C(\ell_s) \Rightarrow C(\ell_t), \quad I(\ell_t) \Rightarrow I(\ell_s)
\]

\[
\ell_t^{-} \leq \ell_s^{-} \lor \bigwedge(\ell_s^{-}) \leadsto I(\ell_t) \land C(\ell_t) \Rightarrow C(\ell_t)
\]

\[
\ell_t^{-} \leq \ell_s^{-} \lor \bigwedge(\ell_t^{-}) \leadsto I(\ell_t) \Rightarrow C(\ell_t) \lor I(\ell_t)
\]

Figure 8. Translating flows-to constraints over labels to acts-for constraints over label components.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Update rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1 \Rightarrow L_2$</td>
<td>$L_1^{+1} := L_1 \land L_2$</td>
</tr>
<tr>
<td>$L_1 \land p_2 \Rightarrow L_3$</td>
<td>$L_1^{+1} := L_1 \land (p_2 \Rightarrow L_3)$</td>
</tr>
<tr>
<td>$L_1 \Rightarrow L_2 \lor L_3$</td>
<td>$L_1^{+1} := L_1 \land (L_2 \lor L_3)$</td>
</tr>
</tbody>
</table>

Figure 9. Update rules for solving acts-for constraints.

NMIFC, endorsement is needed to make the guard high-integrity. A naive programmer might think to endorse the entire guard, but this (nontransparent) endorsement could still be compiled in a way that lets an untrusted host supply its value. The correct solution is to explicitly endorse guess before declassifying the comparison; since guess is not secret, the endorsement is transparent. The resulting labels correctly force Viaduct to put the comparison on the server.

### 3.2 Label Inference

Checking secure information flow is not enough; for protocol selection, the compiler also needs the labels of all expressions. We present an algorithm to infer these labels.

As in prior work on inferring information flow labels [37, 40], information flow checking reduces to a system of flows-to ($\subseteq$) constraints over label constants and label variables. Type inference collects these premises from Figure 7, and generates fresh label variables for labels that appear in a premise of a rule but not its conclusion (e.g., $pc'$ in the rule for if statements). The inference algorithm finds a label-variable assignment that satisfies all the constraints, if possible.

The algorithm computes the minimum-authority solution, the choice of labels requiring the least amount of confidentiality and integrity for each component. Minimum-authority labels are desirable because higher authority is achieved only through more trust or costly cryptography.

First, we translate the flows-to ($\subseteq$) constraints over labels, which appear in rule premises, to acts-for ($\Rightarrow$) constraints over the underlying label components as shown in Figure 8. Here, $C(\ell)$ and $I(\ell)$ are functions that project the confidentiality and integrity components, respectively, of label $\ell$. These components are constants $p$ when the label is known, and variables $L$ otherwise.

We then adapt the algorithm of Rehof and Mogensen [42] for iteratively solving semilattice constraints. All principal variables are initialized to 1 and unsatisfied constraints are used to update variables repeatedly, until a fixed point is reached.
Viaduct

\[
\begin{array}{l}
\Pi \models e : P \\
\Pi \models a : P \\
\Pi \models \text{endorse } a \text{ from } t : P \\
\Pi \models \text{input}_P h : \text{Local}(h) \\
\Pi \models \text{output } a \text{ to } h : \text{Local}(h) \\
\end{array}
\]

\[
\begin{array}{l}
\Pi \models \text{let } t = e \text{ in } s \\
\Pi \models \text{new } x = D(a_1, \ldots, a_n) \text{ in } s \\
\Pi \models \text{if } a \text{ then } s_1 \text{ else } s_2 \\
\Pi \models \text{break } b \\
\Pi \models \text{skip} \\
\end{array}
\]

**Figure 10.** Rules for the validity of a protocol assignment.

\[
\Pi(s) : 2^\Pi \quad \text{hosts}(\Pi, s) : 2^\Pi
\]

\[
\begin{array}{l}
\Pi(\text{let } t = e \text{ in } s) = \Pi(t) \cup \Pi(s) \\
\Pi(\text{new } x = D(a_1, \ldots, a_n) \text{ in } s) = \Pi(x) \cup \Pi(s) \\
\Pi(\text{if } a \text{ then } s_1 \text{ else } s_2) = \Pi(s_1) \cup \Pi(s_2) \\
\Pi(b : \text{loop } s) = \Pi(s) \\
\Pi(\text{break } b) = \Pi(b : \text{loop } s) \\
\Pi(s_1; s_2) = \Pi(s_1) \cup \Pi(s_2) \\
\Pi(\text{skip}) = \emptyset \\
\end{array}
\]

\[
\text{hosts}(\Pi, s) = \bigcup_{P \in \Pi(s)} \text{hosts}(P)
\]

**Figure 11.** Protocols and hosts involved in the execution of a statement. Here, hosts(P) is the set of hosts that protocol P runs on, which is specified individually for each protocol.

**cost(\Pi, let t = e in s) =**

\[
c_{\text{exec}}(\Pi(t), e) + \sum_{P \in \Pi(\Pi(t), s)} c_{\text{comm}}(\Pi(t), P) + \text{cost}(\Pi, s)
\]

**cost(\Pi, if a then s_1 else s_2) =**

\[
\max(\text{cost}(\Pi, s_1), \text{cost}(\Pi, s_2))
\]

**cost(\Pi, b : loop s) =**

\[
W_{\text{loop}} \times \text{cost}(\Pi, s)
\]

**cost(\Pi, s_1; s_2) =**

\[
\text{cost}(\Pi, s_1) + \text{cost}(\Pi, s_2)
\]

**cost(\Pi, s) = 0 otherwise**

**Figure 12.** Abstract cost model.

Premise ensures that the protocol computing the value of the guard can forward it to all involved hosts. Both premises are trivially satisfied when the guard is a constant expression.

Where necessary, the Viaduct compiler removes these guard visibility constraints by multiplexing [35] conditional statements into straight-line code. This allows, for example, the compilation of conditionals with secret guards that require execution in MPC.

### 4.2 Cost of Protocol Assignments

There can be many valid protocol assignments that securely realize a source program. To select an optimal assignment, Viaduct attributes a cost to each assignment using an abstract cost model, shown in Figure 12. Developers can instantiate the abstract model by modifying the customizable cost estimator, which specifies \( c_{\text{exec}}(P, s) \), the cost of executing statement \( s \) in protocol \( P \); \( c_{\text{comm}}(P_1, P_2) \), the cost of communicating between \( P_1 \) and \( P_2 \); and the global constant \( W_{\text{loop}} \), the number of times a loop is assumed to execute when its iteration count is not statically known.

Our implementation configures \( c_{\text{exec}} \) to assign a small cost to executing “in the clear” and a large cost to the use of cryptography, so the compiler avoids the use of cryptography except when required for security. We also configure the communication cost \( c_{\text{comm}} \) to minimize data movement. For example, a frequently accessed public variable would be replicated on two hosts so that each host has a local copy. Placing the variable only on one of the hosts could reduce storage cost but entails frequently sending its value to the other host.
4.3 Computing an Optimal Protocol Assignment

To compute an optimal protocol assignment given a program \( s \), the Viaduct compiler constructs a constrained optimization problem over the following sets of variables:

- **Assignment variables** \( \alpha \): These represent the protocols that execute let-bindings or declarations.
- **Cost variables** \( \beta \): These represent the cost of executing let-bindings or declarations.
- **Participating host variables** \( y_{i,j} \): These are true if host \( j \) is participating in the execution of a statement \( i \).

The compiler generates a set of constraints \( \{ \phi_1, \ldots, \phi_n \} \) over these assignment, cost, and participating host variables, as well as an expression \( \beta_i \) capturing the cost of \( s \) as in Figure 12. These constraints are drawn from a grammar consisting of logical connectives, an equality predicate between assignment variables and protocols, and an equality predicate between cost variables and cost expressions. The compiler uses an off-the-shelf solver to find a solution for assignment variables \( \alpha \) and participating host variables \( y_{i,j} \) such that all constraints \( \{ \phi_1, \ldots, \phi_n \} \) are satisfied and \( \beta_i \) is minimized.

Given the set of valid protocol assignments \( VA \) for \( s \) such that \( VA = \{ \Pi \mid \Pi \models s \} \), this solution for the assignment variables corresponds to a protocol assignment \( \Pi_{opt} \) such that

\[
\Pi_{opt} = \arg\min_{\Pi \in VA} \text{cost}(\Pi, s).
\]

**Protocol Factory.** To construct the optimization problem, the compiler draws the set of available protocols from the customizable protocol factory. Developers wishing to add new protocols to Viaduct must extend the protocol factory so that the compiler can generate assignments with these protocols during program selection.

The protocol factory defines a function \( \text{viable} : \mathcal{T} \cup \mathcal{X} \rightarrow 2^\beta \) that returns a set of viable protocols that can execute a let-binding or declaration. This allows developers to specify limitations regarding the use of particular protocols. For example, commitment protocols may be unable to compute over commitments. Other protocols may lack support for certain operators.

**Example.** Consider the following source program to be executed by hosts \( a \) and \( b \):

```plaintext
let t₁ = 1 + 1 in let t₂ = t₁ × 2 in skip
```

and the following data from the compiler’s label inference phase and extension points:

1. \( \text{viable}(t₁) = \{ P₁, P₂, P₄ \}, \text{viable}(t₂) = \{ P₁, P₂ \} \)
2. \( \mathbb{L}(P₁) \Rightarrow \mathbb{L}(t₁), \mathbb{L}(P₂) \Rightarrow \mathbb{L}(t₁), \mathbb{L}(P₄) \neq \mathbb{L}(t₁) \)
3. \( \mathbb{L}(P₁) \Rightarrow \mathbb{L}(t₂), \mathbb{L}(P₂) \Rightarrow \mathbb{L}(t₂) \)
4. \( \text{hosts}(P₁) = \{ a \}, \text{hosts}(P₂) = \{ b \}, \text{hosts}(P₄) = \{ a, b \} \)
5. \( \text{cexec}(P₁, \_1) = 5, \text{cexec}(P₂, \_1) = 5, \text{cexec}(P₃, \_1) = 3 \)
6. \( \text{comm}(P₁, P₁) = 0, \text{comm}(P₃, P₂) = 2 \)
7. \( \text{cexec}(P₁, P₁), \text{comm}(P₃, P₁) \)
8. \( \text{comm}(P₃, P₂), \text{comm}(P₁, P₂) \)

Then the compiler constructs the problem of minimizing cost \( \beta₁ + \beta₂ \) while satisfying the following constraints:

\[
\begin{align*}
\alpha₁ &= P₁ \lor \alpha₂ = P₃ \\
\alpha₁ &= P₁ \rightarrow (y₁,a \land \neg y₁,b \land \beta₁ = 5) \\
\alpha₁ &= P₃ \rightarrow (y₁,a \land y₁,b \land \beta₁ = 3) \\
\alpha₂ &= P₁ \rightarrow (y₂,a \land \neg y₂,b \land \alpha₁ \neq P₃ \land (\alpha₁ = P₁ \rightarrow \beta₂ = 5 + 0)) \\
\alpha₂ &= P₂ \rightarrow (\neg y₂,a \land y₂,b \land \alpha₁ \neq P₃ \land (\alpha₁ = P₃ \rightarrow \beta₂ = 3 + 2))
\end{align*}
\]

Note that \( \alpha₁, \beta₁, \gamma₁,a \), and \( \gamma₁,b \) are variables associated with \( t₁ \) while \( \alpha₂, \beta₂ \) and \( \gamma₂,a, \gamma₂,b \) are variables associated with \( t₂ \). The first constraint bounds the possible values of assignment variables \( \alpha₁ \) and \( \alpha₂ \) and is generated from the viable protocols returned by the protocol factory. Viable protocols that do not meet authority requirements are filtered out, so \( P₁ \) is not a possible value for \( \alpha₁ \). The rest of the constraints describe the relationship between protocol assignments, participating hosts, possible protocol compositions, and cost. From this optimization problem the compiler then computes the optimal assignment \( \Pi_{opt} \) where \( \Pi_{opt}(t₁) = P₁ \) and \( \Pi_{opt}(t₂) = P₂ \).

5 Viaduct Runtime

Once it has computed a protocol assignment, the Viaduct compiler outputs a program where every let-binding and assignable declaration is annotated with the protocol that will execute it. This annotated program can be executed by the Viaduct runtime, which consists of an extensible interpreter that interacts with a set of protocol back ends, each of which implement a set of protocols. The interface for protocol back ends is straightforward: back ends must implement methods to execute let-bindings and assignable declarations, and methods to communicate with other protocol back ends.

Each host runs a copy of the interpreter with the annotated program as input. For each statement, the interpreter checks whether the host participates in its execution, as defined by hosts(\( \Pi \_ \_ ) \) if not, the statement is treated like skip. If a host participates in executing a let-binding or a declaration, the interpreter calls the back end for the protocol assigned to the statement. To execute a conditional, the host retrieves the cleartext value of the guard from the protocol back end that stores it, and executes the appropriate branch. The validity rules for protocol assignments ensure the host is allowed to see the cleartext value, and that it is able to retrieve it.

5.1 Protocol Composition

The protocol back end executing a let-binding must send the computed value to back ends executing statements that read the bound temporary. How one back end sends a value to another depends on the protocols involved. For example, a statement executed in Replicated(\( h₁, h₂ \)) reading a temporary computed in SH-MPC(\( h₁, h₂ \)) corresponds to executing

\[\text{Note that participating host variables are unused here, but in general they encode the guard visibility constraint for conditionals.}\]
Viaduct

<table>
<thead>
<tr>
<th>Sending protocol (s)</th>
<th>Receiving protocol (r)</th>
<th>Communication</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local($h_1$)</td>
<td>SH-MPC($h_1$, $h_2$)</td>
<td>($s$, $h_1$) $\rightarrow$ ($r$, $h_1$)</td>
<td>create input gate for MPC circuit</td>
</tr>
<tr>
<td>Local($h_2$)</td>
<td>Commitment($h_p$, $h_v$)</td>
<td>($s$, $h_p$) $\rightarrow$ ($r$, $h_p$)</td>
<td>create commitment</td>
</tr>
<tr>
<td>Replicated($h_1$, $h_2$)</td>
<td>SH-MPC($h_1$, $h_2$)</td>
<td>($s$, $h_1$) $\rightarrow$ ($r$, $h_1$), ($s$, $h_2$) $\rightarrow$ ($r$, $h_2$)</td>
<td>read replicated data</td>
</tr>
<tr>
<td>SH-MPC($h_1$, $h_2$)</td>
<td>Replicated($h_1$, $h_2$)</td>
<td>($s$, $h_1$) $\rightarrow$ ($r$, $h_1$), ($s$, $h_2$) $\rightarrow$ ($r$, $h_2$)</td>
<td>execute circuit and reveal output</td>
</tr>
<tr>
<td>Commitment($h_p$, $h_v$)</td>
<td>Local($h_v$)</td>
<td>($s$, $h_p$) $\rightarrow$ ($r$, $h_v$), ($s$, $h_v$) $\rightarrow$ ($r$, $h_v$)</td>
<td>open commitment</td>
</tr>
<tr>
<td>ZKP($h_p$, $h_v$)</td>
<td>Local($h_v$)</td>
<td>($s$, $h_v$) $\rightarrow$ ($r$, $h_v$)</td>
<td>send result and proof to verifier</td>
</tr>
</tbody>
</table>

Figure 13. Selected examples of protocol composition. The $ct$ port of various protocols stands for cleartext input; the $in$ port of the MPC protocol represents secret input from a host; the $cc$ port of the Commitment protocol represents creating a commitment; the $occ$ and $ohc$ ports of the Local protocol respectively represent receiving the cleartext value of an opened commitment and the commitment itself.

We implemented the Viaduct compiler in about 20 KLoC of Kotlin code, which includes code for the parser, the label constraint solver, protocol selection, and the runtime system. The code written against the compiler’s extension points—the protocol factory, the protocol composer, the cost estimator, and the protocol back ends—runs to about 4 KLoC. Viaduct uses the Z3 SMT solver [14] to solve the optimization problem generated during protocol selection.

We implemented four protocol back ends for Viaduct:

**Local/Replicated.** The cleartext back end executes code in Local and Replicated protocols. It maintains a store for objects that directly represent the temporaries and assignables of the source program. Computations performed by the cleartext back end are executed directly.

**SH-MPC.** This back end links Viaduct to ABY, a library for two-party semi-honest MPC [16]. It maintains a store of gate objects that represent circuit components executed by ABY. Computations performed by the back end build gate objects that represent the operation performed (e.g., an addition in the source program creates an ADD gate).

The ABY framework supports execution of circuits in three different schemes—arithmetic sharing, boolean sharing, and
Yao’s garbled circuits—as well as conversions between these, allowing for execution of mixed-protocol circuits. Viaduct represents each scheme as a separate protocol, but all three are implemented by a single back end. To generate efficient mixed circuits, we follow Demmler et al. [16] and Ishaq et al. [28] and estimate inputs to the cost estimator by measuring execution time of individual operations under a particular scheme and conversions between schemes. We perform measurements for two settings: low-latency, high-bandwidth (LAN), and high-latency, low-bandwidth (WAN). Thus the cost estimator has two modes, each of which optimizes compiled programs for a specific network environment.

**Commitment.** This back end manages commitments, implemented using SHA-256 hashes of data along with a nonce. The back end for the commitment creator maintains a store of cleartext values along with metadata for commitments. The back end for the commitment receiver maintains the set of commitments, as hashes. The commitment back end cannot support computation.

**ZKP.** This back end links to libsnark [1], a library for zkSNARKs (zero-knowledge Succinct Non-interactive Arguments of Knowledge). This back end maintains a store of circuit gate objects. The prover and verifier both manage cleartext values for the public inputs to the protocol, while only the prover manages cleartext values for the secret inputs. To ensure the prover cannot modify secret inputs mid-execution, all secret inputs are “committed” by sending their hash to the verifier. All proofs that use a secret input then include a clause that equates the input to the pre-image of the hash held by the verifier.

The libsnark library requires proving and verifying keys to be generated for each unique circuit before the protocol is executed. The current prototype requires a “dummy” run of the compiled program to generate these keys.

7 Evaluation

To evaluate Viaduct, we address these research questions:

- **RQ1:** Is Viaduct expressive enough?
- **RQ2:** Is its compilation performance acceptable?
- **RQ3:** Does it generate efficient distributed programs?
- **RQ4:** How much does label inference reduce the annotation burden for programmers?
- **RQ5:** What is the overhead of the runtime system?

Experiments used Dell OptiPlex 7050 machines with an 8-core Intel Core i7 7th Gen CPU and 16 GB of RAM. Note that for experiments involving time measurements (RQ2, RQ3, RQ5), the numbers reported are over 5 trials and the relative standard error is at most 6% of the sample mean.

Existing work such as Büscher et al. [7] and Ishaq et al. [28] focus on optimizing mixed circuits for ABY specifically, and as such these employ more sophisticated reasoning about cost for ABY circuits. We consider it future work to incorporate such techniques into Viaduct.

**RQ1 - Expressiveness.** Figure 14 shows the benchmarks used for the experiments and the cryptography synthesized by Viaduct for each benchmark. Several are from prior work, rewritten in the Viaduct source language. Host configurations are either semi-honest, as in Figure 2, where hosts A and B trust each other for integrity, mutually distrusting as in Figure 3; or are “hybrid” configurations where A and B trust each other but host C is trusted by neither.

Our benchmarks show that Viaduct can compile programs whose security demands a variety of cryptographic mechanisms. With hybrid configurations (interval, bet), Viaduct combines MPC and ZKP to implement different components of a single distributed program. Code for selected benchmarks can be found in the supplemental technical report [2].

**RQ2 - Scalability of Compilation.** The two main phases of the Viaduct compiler are label inference and protocol selection. Our benchmarks indicate that the overhead of label inference is negligible: at most several hundred milliseconds. As seen in Figure 14, the overhead for protocol selection is more significant, but still on the order of several seconds for most benchmarks. The longest running benchmark, k-means, performs most of its computations in MPC. In this case, it may be harder to converge to the optimal solution since the solver generates a large mixed circuit, choosing between the three MPC schemes supported by ABY.

**RQ3 - Cost of Compiled Programs.** To show that Viaduct can compile efficient distributed programs, we chose a subset of our benchmarks requiring the use of MPC and compared the execution of optimal programs generated by Viaduct—for each benchmark, one optimized for local area networks (LAN) and another for wide area networks (WAN)—with naive protocol assignments that perform all computation in MPC. The naive ABY assignments use either boolean sharing or Yao garbled circuits, since arithmetic sharing can only perform arithmetic operations. We measured executions in a 1 Gbps LAN and simulated WAN (100 Mbps bandwidth and 50 ms latency). We configured ABY to use 32-bit integers and set its security parameter to 128 bits.

Figure 15 summarizes our results. For some benchmarks (HHI score, hist. millionaires, median, two-round bidding), computation can be securely moved from MPC to cleartext protocols, making execution much more efficient. Even for benchmarks that require computations to be almost entirely in MPC (bio. match, k-means), Viaduct chooses efficient mixed circuits that perform much better than the naive assignments entirely in boolean sharing or Yao circuits. Viaduct replicates the result in Büscher et al. [7] (which specifically targets the ABY framework) in choosing a mix of arithmetic and Yao circuits as optimal assignments for the two benchmarks from that paper, with the exception of the k-means benchmark in the WAN setting.
### Figure 14. Benchmark programs. **Protocols** give the protocols used in the compiled program for either the LAN or WAN setting. Legend for protocols used: A, B, Y—ABY arithmetic/boolean/Yao sharing; C—Commitment; L—Local; R—Replicated; Z—ZKP. **Ann** gives the minimum number of label annotations needed to write the program. **Selection** gives the number of symbolic variables and run time in seconds for protocol selection, averaged across five runs.

### Table 1

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Protocols</th>
<th>LoC</th>
<th>Ann</th>
<th>Vars</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>battleship</td>
<td>model of the board game</td>
<td>RZ / RZ</td>
<td>79</td>
<td>12</td>
<td>1022</td>
<td>1.0</td>
</tr>
<tr>
<td>bet</td>
<td>C bets who wins hist. millionaires b/w A &amp; B</td>
<td>CLRY / CLRY</td>
<td>79</td>
<td>7</td>
<td>1022</td>
<td>1.0</td>
</tr>
<tr>
<td>biometric match</td>
<td>min distance b/w sample &amp; database (from [7])</td>
<td>ALRY / ALRY</td>
<td>40</td>
<td>8</td>
<td>708</td>
<td>2.0</td>
</tr>
<tr>
<td>guessing game</td>
<td>same as in fig. 3</td>
<td>RZ / RZ</td>
<td>16</td>
<td>6</td>
<td>193</td>
<td>0.4</td>
</tr>
<tr>
<td>HHI score</td>
<td>compute market concentration index (from [46])</td>
<td>ALRY / LRY</td>
<td>22</td>
<td>3</td>
<td>285</td>
<td>1.1</td>
</tr>
<tr>
<td>historical millionaires</td>
<td>same as in fig. 2 but with arrays</td>
<td>LRY / LRY</td>
<td>17</td>
<td>3</td>
<td>187</td>
<td>0.7</td>
</tr>
<tr>
<td>interval</td>
<td>A &amp; B compute interval of combined points, C attests point is in interval</td>
<td>RYZ / RYZ</td>
<td>45</td>
<td>9</td>
<td>660</td>
<td>2.8</td>
</tr>
<tr>
<td>k-means</td>
<td>cluster secret points from A &amp; B (from [7])</td>
<td>ARY / RY</td>
<td>82</td>
<td>3</td>
<td>1684</td>
<td>7.9</td>
</tr>
<tr>
<td>k-means (unrolled)</td>
<td>k-means w/ 3 unrolled iterations</td>
<td>ARY / RY</td>
<td>174</td>
<td>3</td>
<td>3629</td>
<td>29.0</td>
</tr>
<tr>
<td>median</td>
<td>compute median of A &amp; B’s lists (from [30])</td>
<td>RY / RY</td>
<td>36</td>
<td>6</td>
<td>386</td>
<td>1.0</td>
</tr>
<tr>
<td>rock-paper-scissors</td>
<td>A &amp; B commit to moves then reveal</td>
<td>CR / CR</td>
<td>56</td>
<td>6</td>
<td>741</td>
<td>1.0</td>
</tr>
<tr>
<td>two-round bidding</td>
<td>A &amp; B bid for a list of items</td>
<td>LRY / LRY</td>
<td>34</td>
<td>4</td>
<td>575</td>
<td>1.7</td>
</tr>
</tbody>
</table>

### Figure 15. Run time (in seconds) and communication (in MB) of select benchmark programs, averaged across five runs. **Bool** and **Yao** are naive assignments using boolean sharing and Yao sharing respectively to execute MPC computations. **Opt-LAN** and **Opt-WAN** are optimal assignments generated by Viaduct for the LAN and WAN setting respectively. Optimal time and communication for a benchmark and execution setting pair are in **bold**.

### Table 2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LAN Time</th>
<th>Slowdown</th>
<th>WAN Time</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>bio. match</td>
<td>3.6</td>
<td>95.9</td>
<td>56.0</td>
<td></td>
</tr>
<tr>
<td>HHI score</td>
<td>0.8</td>
<td>9.7</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>hist. million.</td>
<td>1.0</td>
<td>90.6</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td>k-means</td>
<td>56.5</td>
<td>696.1</td>
<td>1273.1</td>
<td></td>
</tr>
<tr>
<td>median</td>
<td>11.5</td>
<td>1098.7</td>
<td>197.1</td>
<td></td>
</tr>
<tr>
<td>2-R bidding</td>
<td>17.3</td>
<td>184.7</td>
<td>233.0</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 16. Run time (in seconds) of LAN-optimized benchmarks hand-written to use ABY directly and the slowdown of running the same benchmarks through the Viaduct runtime in LAN and WAN settings.

### RQ4 - Annotation Burden of Security Labels. Security-typed languages add some annotation burden when writing programs. In practice, labels on host declarations and downgrading operations suffice to specify intended security policies in Viaduct programs. To substantiate this claim, we created two versions of each benchmark program. In one, every variable has a label annotation; in the other, “erased” version, all such labels are omitted.

For all benchmarks, Viaduct generates the same compiled program for the fully labeled and the erased versions. Although the inferred labels for the erased programs are not exactly the same as in their manually labeled counterparts, the differences do not affect the protocols chosen. The **Ann** column in Figure 14 counts label annotations needed to write the program: effectively, the number of downgrades
plus the number of host declarations, each of which need an authority label. The table shows that the annotation burden is low: most benchmarks need only a few label annotations.

8 Related Work
Compilation to Cryptographic Protocols. The idea of compiling a high-level program to a cryptographic protocol has been explored in the context of multiparty computation [27] (e.g., Fairplay [35], SCVM [33], ObliVM [34], ObliV C [50], Wysteria [41], HyCC [7], SCALE-MAMBA [3]), and that of zero-knowledge proofs (e.g., Pinocchio [38], Gennaro et al. [13], Buffet [47], jxSNARK [32]). Earlier work is generally limited to the domain of a particular fixed cryptographic task (e.g., MPC or ZKP); Viaduct’s novelty is synthesizing efficient protocols across cryptographic tasks. Like SCVM [33], Viaduct can synthesize “hybrid” programs that perform computations locally, replicated between hosts, or under MPC. This is impossible in the simple two-point label model that many MPC compilers [3, 34] use, which only distinguish between public (low) and secret (high) information. Viaduct also does not fix the number of hosts in a program (unlike [33–35]), nor fix compiling programs only under a semi-honest or malicious setting (unlike [32–34, 38, 41, 47]).

Program Partitioning. Another line of related work [19, 20, 52, 53] describes distributed computations using sequential programs and captures security requirements using information-flow labels. The Jif/split compiler [52, 53] synthesizes simple cryptographic primitives such as cryptographic commitments to satisfy security constraints that would otherwise be impossible without relying on trusted principals. Unlike Viaduct, Jif/split is not extensible to new protocols. Later work [19, 20] proves computational soundness for a similar system under a strong attacker that controls the network and some of the hosts. However, this work does not support replicating computations (only data replication is supported), or the other protocols that Viaduct supports.

9 Conclusion
The prototype implementation of the Viaduct compiler compiles high-level, security-typed programs into efficient distributed programs that employ a variety cryptographic mechanisms to ensure security. And the compiler is agnostic to the set of available protocols, making it easily extensible.

Promising avenues for future work remain. The label model could be extended with availability policies [54], guiding selection of fault-tolerant protocols like quorum replication [55] and MPC with guaranteed output delivery [26]. A more full-fledged implementation of Viaduct could support executing code on trusted execution environments like hardware enclaves [25, 29, 36], the use of special-purpose protocols like private set intersection [12, 39] and Oblivious RAM [45], and the incorporation of a more detailed and accurate cost model [28].

Finally, a full correctness proof for the Viaduct compiler would be a significant research achievement, bridging security notions defined by the programming-languages and cryptography communities. One can see Viaduct source programs as ideal functionalities and the distributed programs generated by the compiler as hybrid protocols using ideal functionalities implemented by cryptographic mechanisms. The conjectured correctness statement for Viaduct is a simulation proof in the Universal Composability (UC) framework [9], relating a Viaduct source program to the distributed implementation generated by the compiler.

Acknowledgments
This work was supported by the National Science Foundation under grant CNS-1704788, and by the IARPA HECTOR program under a subcontract from IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any of these funders.

We thank our shepherd Mike Hicks and our anonymous reviewers for their insightful suggestions. We also thank Alexa van Hattum, Tobias Kappe, Ralph Recto, and Drew Zagieboylo for feedback during the drafting of this paper.
References


A Termination and Optimality of the Label Inference Algorithm

In this section, we prove that the iterative analysis we use for label inference always terminates and computes the minimum-authority solution. First, we construct the $\rightarrow$ operator over the lattice of principals, which occurs in the update rules.

A.1 Constructing the $\rightarrow$ Operator

We show that any free distributive lattice, like our lattice of principals, is a Heyting algebra, and thus the $\rightarrow$ operator we use in our label inference algorithm (§3.2) is well-defined. While this is a standard result in algebra, we believe it is illuminating to see the actual construction of $\rightarrow$, as we use its concrete value to compute minimum-authority labels.

A.1.1 Free Distributive Lattices. Let $P$ be an arbitrary set. The standard construction for the free distributive lattice over $P$ takes finite sets of finite subsets of $P$ as elements, which we write as

$\{A_i\}_{i\in[n]}$ (where $A_i \subseteq P$).

An element of this form is interpreted as a join of meets, that is, $\{A_i\}_{i\in[n]}$ intuitively stands for

$\left(\bigwedge A_1\right) \lor \ldots \lor \left(\bigwedge A_n\right)$.

In addition to every $A_i$ being finite, we require that there is no $A_i$ and $A_j$ such that $A_i \subseteq A_j$ for $i \neq j$ since this makes $A_j$ redundant per our interpretation (i.e. $\left(\bigwedge A_i\right) \lor \left(\bigwedge A_j\right) = \left(\bigwedge A_i\right)$). We assume all such components are dropped implicitly.

Define

$\{A_i\}_{i\in[n]} \lor \{B_j\}_{j\in[m]} = \{A_i\}_{i\in[n]} \cup \{B_j\}_{j\in[m]}$

and

$\{A_i\}_{i\in[n]} \land \{B_j\}_{j\in[m]} = \{A_i \cup B_j \mid i \in [n], j \in [m]\}$.

It is straightforward to verify that these definitions satisfy the properties for being the join and the meet, respectively. It is also easy to see that

$0 = \{\} \quad \text{and} \quad 1 = \{\{\}\}.$

Finally, ordering can be derived in the standard way for distributive lattices:

$A \leq B \iff A \lor B = B$.

We find it useful to have a more direct definition, which we can derive by expanding the previous definition:

$\{A_i\}_{i\in[n]} \leq \{B_j\}_{j\in[m]} \iff \forall i \in [n].\exists j \in [m].B_j \subseteq A_i$.

A.1.2 Heyting Algebras. A Heyting algebra is a bounded distributive lattice where every inequality of the form

$A \land X \leq B$

has a greatest solution. This solution is named $A \rightarrow B$ to appeal to logical intuition as $A \rightarrow B$ is the weakest (i.e. the greatest) proposition such that $A \land (A \rightarrow B)$ logically implies $B$. We show that every free distributive lattice forms a Heyting algebra. Define

$\{A_i\}_{i\in[n]} \rightarrow \{B_j\}_{j\in[m]} = \bigwedge_{i\in[n]} \{B_j \setminus A_i \mid j \in [m]\}$.

First, we claim this is in fact a solution to the above inequality, that is,

$\{A_i\}_{i\in[n]} \land \bigwedge_{i\in[n]} \{B_j \setminus A_i \mid j \in [m]\} \leq \{B_j\}_{j\in[m]}$.

Proof: By applying the definition of $\land$ repeatedly ($i + 1$ times), we can rewrite the left-hand side as

$\{A_i \cup (B_{j_1} \setminus A_1) \cup \ldots \cup (B_{j_n} \setminus A_n) \mid i \in [n], j_1, \ldots, j_n \in [m]\}$.

Using the direct definition of $\leq$ from before, it suffices to show that there exists $j \in [m]$ such that

$B_j \subseteq A_i \cup (B_{j_1} \setminus A_i) \cup \ldots \cup (B_{j_n} \setminus A_i)$

for all $i, j_1, \ldots, j_n$. Picking $j = j_i$, we get

$B_{j_i} \subseteq A_i \cup (B_{j_i} \setminus A_i) \subseteq A_i \cup (B_{j_i} \setminus A_i) \cup \ldots \cup (B_{j_n} \setminus A_i)$.
Next, we need to prove that this solution is the greatest. Assume there is an $X$ such that $A \land X \leq B$ where $A = \{A_i\}_{i \in [n]}$, $B = \{B_j\}_{j \in [m]}$, and $X = \{X_k\}_{k \in [o]}$. Our goal is to show
\[
\{X_k\}_{k \in [o]} \leq \bigwedge_{i \in [n]} \{B_j \setminus A_i \mid j \in [m]\}.
\]

**Proof.** Using the universal property of $\land$ and the direct definition of $\leq$ from before, it is sufficient to prove
\[
\forall i \in [n], k \in [o]. \exists j \in [m]. B_j \setminus A_i \subseteq X_k.
\]

Let $i$ and $k$ be arbitrary. Since $\{A_i\} \leq A$ and $\{X_k\} \leq X$, we know
\[
\{A_i\} \land \{X_k\} \leq A \land X \leq B \implies \exists j \in [m]. B_j \subseteq A_i \cup X_k
\]
\[
\implies \exists j. B_j \setminus A_i \subseteq X_k.
\]

\[\square\]

### A.2 Termination and Optimality

It is well-known that iterative analysis always terminates given that the function defined by the update rules is monotone, and that the lattice over which the algorithm runs is of finite height [31]. Because the update rules take the meet of the current solution with some other lattice element, it is immediate that the function is monotone. Because it is the free distributive lattice, all elements of the principal lattice can be represented in normal form as a join of meets of atomic principals, and thus is of finite size when the set of atomic principals is finite. Thus the principal lattice is of finite height as long as it is generated from a finite set of atomic principals. We know any program can only reference a finite set of unique atomic principals in its text since any program has a finite set of labels in its text, and each label can only mention a finite set of atomic principals. Thus for any program, the principal lattice is of finite height.

Finally, we show that the algorithm computes the optimal (minimum-authority) solution. It is also well-known by appeal to Kleene’s fixed-point theorem that iterative analysis computes the greatest-fixpoint solution of a monotone function. Thus to prove optimality it is sufficient to show that any solution to the constraints must lower-bound the current solution computed from the update rules, and thus must lower-bound the greatest-fixpoint solution computed by the algorithm.

**Proof.** We prove the statement by induction over the number of iterations performed by iterative analysis. The base case is immediate since all principal variables are initialized to $1$, the top of the principal lattice.

To prove the inductive case, we perform a case analysis over the update rules:

**Case 1:** $L_i^{i+1} := L_i^1 \land L_i^2$. For solution including $L_i^1$ and $L_i^2$ such that $L_i^1 \Rightarrow L_i^2$, we know by the inductive hypothesis $L_i^1 \Rightarrow L_i^1$ and $L_i^2 \Rightarrow L_i^2$, and thus $L_i^2 \Rightarrow L_i^1$ by transitivity. Since $\land$ is the greatest lower bound, $L_i^1 \Rightarrow L_i^1 \land L_i^2 = L_i^{i+1}$, as needed.

**Case 2:** $L_i^{i+1} := L_i^1 \land (p \rightarrow L_i^1)$. For solution including $L_i^1$ and $L_i^1$ such that $L_i^1 \land p \Rightarrow L_i^1$, we know by the inductive hypothesis $L_i^1 \Rightarrow L_i^1$ and $L_i^1 \Rightarrow L_i^1$, and thus $L_i^1 \land p \Rightarrow L_i^1$ by transitivity. By definition we know $p \rightarrow L_i^1$ is the greatest principal $p$ such that $p \land p \Rightarrow L_i^1$, so $L_i^1 \Rightarrow p \rightarrow L_i^1$. Since $\land$ is the greatest lower bound, $L_i^1 \Rightarrow L_i^1 \land (p \rightarrow L_i^1) = L_i^{i+1}$ as needed.

**Case 3:** $L_i^{i+1} := L_i^1 \land (L_i^2 \lor L_i^3)$. For solution including $L_i^1$, $L_i^2$, and $L_i^3$ such that $L_i^1 \Rightarrow (L_i^2 \lor L_i^3)$, we know by the inductive hypothesis that $L_i^1 \Rightarrow L_i^1$ and $L_i^2 \Rightarrow L_i^1$ and $L_i^3 \Rightarrow L_i^1$. Thus $L_i^1 \Rightarrow L_i^2 \lor L_i^3 \Rightarrow L_i^1$ and since $\land$ is the greatest lower bound, $L_i^1 \Rightarrow L_i^1 \land (L_i^2 \lor L_i^3) = L_i^{i+1}$ as needed.

\[\square\]

### B Selected benchmarks

The following sections have the Viaduct source code for a select number of benchmarks and a description of the distributed programs that the compiler generates for each.

For the benchmarks used in RQ5, we also include the Kotlin code for the “bare ABY” programs with which we compared the performance of Viaduct compiled programs. The programs use the Kotlin JNI shim to ABY that the Viaduct compiler uses for its ABY back end. The Kotlin code for the most part uses the ABY API directly using the ABYParty class; the only code that is specific to Viaduct is ABYCircuitBuilder, which is a class that contains references to the arithmetic, and boolean, and Yao circuit objects used to build gates; and Host, which is a wrapper to the String class that contains the name of the current host.

Participating hosts each run a copy of the Kotlin program, so the code uses the ABY API builds the circuit for both hosts (named alice and bob by convention). In some cases the code is the same for both hosts; in other cases the code slightly differs (e.g. alice builds an IN gate while bob build a DummyIN gate), which case the code cases on which is the current host (supplied by the host parameter).
Viaduct

B.1 Battleship

This benchmark runs a game of battleship between Alice and Bob: each player maintains a set of ships located on a map, and then take turns attacking locations where they think an enemy ship resides. Unlike the original board game, in this version the board is one-dimensional and each ship is only 1 unit long.

To execute this program, each player provides the coordinates of their ships as input, which is stored in a private array (Lines 8–11). Then the players execute a cheating detection routine (Lines 20–30): each player reveals to the other player that their ships are not placed in the same location. In the compiled distributed program, this routine is implemented with each player sending zero-knowledge proofs to attest that the locations for each pair of their ships are not equal. A zero-knowledge proof is required here to prevent leaking the locations of the ships.

Alice and Bob then take turns attacking coordinates where they think an enemy ship is located, until one of them sinks all of the ships of the other. On Alice’s turn, she takes a location to attack as input (line 43) and sends this location to Bob, who then sends zero-knowledge proofs attesting whether Alice has sunk one of his battleships (Lines 46–52). Again, zero-knowledge proofs are required here to prevent leaking the locations of ships. Bob’s turn is symmetric to Alice’s.

```java
host alice : {A}
host bob : {B}

// load inputs into endorsed arrays,
// so that they cannot be modified further
val aships = Array[int]({A ∧ B}<-)(5);
val bships = Array[int]({B ∧ A}<-)(5);
for (var i: int = 0; i < 5; i+=1) {
    aships[i] = endorse (input int from alice) from {A};
    bships[i] = endorse (input int from bob) from {B};
}

var awins : bool = false;

// if someone put multiple battleships in the same cell,
// they automatically lose
var acheated : bool = false;
var bcheated : bool = false;
for (var j: int = 0; j < 5 ∧ !acheated ∧ !bcheated; j += 1) {
    for (var k: int = j + 1; k < 5 ∧ !acheated ∧ !bcheated; k += 1) {
        if (declassify (aships[j] == aships[k]) to {A ∧ B}) {
            acheated = true;
        }
    }
}
if (!acheated ∧ !bcheated) {
    var ascore : int = 0;
    var bscore : int = 0;
    var playing : bool = true;
    var aturn : bool = true;
    // keep playing until someone sinks all the other person's battleships
    while (playing) {
```
if (aturn) {
    val amove : int{A ⊓ B->} =
        declassify (input int from alice) to (A ⊓ B->);
    var amove_trusted: int(A ⊓ B) = endorse amove from {A ⊓ B->};
    var ahit: bool(A ⊓ B) = false;
    for (var aj: int{A ⊓ B} = 0; aj < 5; aj += 1) {
        if (declassify (bships[aj] == amove_trusted) to (A ⊓ B)) {
            ascore += 1;
            bships[aj] = 0;
            ahit = true;
        }
    }
    output ahit to alice;
    output ahit to bob;
    aturn = false;
} else {
    var bmove: int{B ⊓ A->} =
        declassify (input int from bob) to (B ⊓ A->);
    val bmove_trusted: int(A ⊓ B) = endorse bmove from (B ⊓ A->);
    var bhit: bool(A ⊓ B) = false;
    for (var bj: int{A ⊓ B} = 0; bj < 5; bj += 1) {
        if (declassify (aships[bj] == bmove_trusted) to (A ⊓ B)) {
            bscore += 1;
            aships[bj] = 0;
            bhit = true;
        }
    }
    output bhit to alice;
    output bhit to bob;
    aturn = true;
}

playing = ascore < 5 ∧ bscore < 5;

awins = ascore == 5;
output awins to alice;
output awins to bob;
} else {
    output bcheated to alice;
    output bcheated to bob;
}

B.2 Biometric Matching

This benchmark computes the minimum Euclidean distance of Bob’s sample to some region in Alice’s database, a common routine in bioinformatics. The Euclidean distance is computed by the match function, which takes as input two points in Alice’s database (db1, db2) and Bob’s sample (s1, s2) and returns the Euclidean distance between these, given as the out parameter res. Note that the labels for the formal parameters of match are upper-bounds; in the Viaduct source language, the concrete label of the arguments at a call site can be referenced in the body of a function by using the parameter name corresponding to the argument, as seen in the labels for dist1 and dist2 (Lines 8–9).
In the compiled implementation generated by Viaduct, Alice and Bob store their respective database and samples locally and then use an MPC protocol to compute the minimum Euclidean distance.

```kotlin
fun match(
    db1: Int, db2: Int, s1: Int, s2: Int,
    res: out Int
) {
    val dist1 = db1 - s1;
    val dist2 = db2 - s2;
    out res = (dist1 * dist1) + (dist2 * dist2);
}

for (var i: Int = 0; i < n * d; i += 1) {
    a_db[i] = input Int from alice;
}

for (var i: Int = 0; i < d; i += 1) {
    b_sample[i] = input Int from bob;
}

match(a_db[0], a_db[1], b_sample[0], b_sample[1], val init_min);
var min_dist = init_min;

for (var i: Int = 0; i < n; i += 1) {
    match(a_db[(i * d)], a_db[(i * d) + 1], b_sample[0], b_sample[1], val dist);
    if (dist < min_dist) {
        min_dist = dist;
    }
}

val result = declassify min_dist to {A \cap B};
output result to alice;
output result to bob;
```

The program is compiled to one semantically equivalent to the Kotlin program below that uses ABY directly.

```kotlin
fun match_alice(db1: Int, db2: Int): Share {
    val tmp = builder.arithCircuit.putINGate(db1.toBigInteger(), BITLEN, builder.role)
    val tmp1 = builder.arithCircuit.putDummyINGate(BITLEN)
    val dist1 = builder.arithCircuit.putSUBGate(tmp, tmp1)
    val tmp3 = builder.arithCircuit.putINGate(db2.toBigInteger(), BITLEN, builder.role)
    val tmp4 = builder.arithCircuit.putDummyINGate(BITLEN)
    val dist2 = builder.arithCircuit.putSUBGate(tmp3, tmp4)
    val tmp8 = builder.arithCircuit.putMULGate(dist1, dist1)
```
val tmp11 = builder.arithCircuit.putMULGate(dist2, dist2)
val tmp12 = builder.arithCircuit.putADDGate(tmp8, tmp11)
return builder.yaoCircuit.putA2YGate(tmp12)
}

fun match_bob(s1: Int, s2: Int): Share {
val tmp = builder.arithCircuit.putDummyINGate(BITLEN)
val tmp1 = builder.arithCircuit.putINGate(s1.toBigInteger(), BITLEN, builder.role)
val dist1 = builder.arithCircuit.putSUBGate(tmp, tmp1)
val tmp3 = builder.arithCircuit.putDummyINGate(BITLEN)
val tmp4 = builder.arithCircuit.putINGate(s2.toBigInteger(), BITLEN, builder.role)
val dist2 = builder.arithCircuit.putSUBGate(tmp3, tmp4)
val tmp8 = builder.arithCircuit.putMULGate(dist1, dist1)
val tmp11 = builder.arithCircuit.putMULGate(dist2, dist2)
val tmp12 = builder.arithCircuit.putADDGate(tmp8, tmp11)
return builder.yaoCircuit.putA2YGate(tmp12)
}

fun benchLANBiomatch(host: Host, aby: ABYParty, builder: ABYCircuitBuilder) {
val n = 500
val d = 4

when (host) {
  'alice' => {
    val a_db = Array<Int>(n * d) { 0 }
    var i = 0
    while (i < n * d) {
      a_db[i] = input.nextInt()
      i += 1
    }

    var min_dist = match_alice(a_db[0], a_db[1])
    var i_2 = 0
    while (i_2 < n) {
      val db1 = a_db[i_2 * d]
      val db2 = a_db[(i_2 * d) + 1]
      val dist = match_alice(db1, db2)
      val tmp50 = builder.yaoCircuit.putGTGate(min_dist, dist)
      val mux = builder.yaoCircuit.putMUXGate(dist, min_dist, tmp50)
      min_dist = mux
      i_2 += 1
    }

    val out = builder.yaoCircuit.putOUTGate(min_dist, Role.ALL)
    executeABYCircuit(aby)
    println(out.clearValue32.toInt())
  }

  'bob' => {
    val b_sample = Array<Int>(d) { 0 }
    var i = 0

    ...
while (i < d) {
    b_sample[i] = input.nextInt()
    i += 1
}

var min_dist = match_bob(b_sample[0], b_sample[1])
var i_2 = 0
while (i_2 < n) {
    val s1 = b_sample[0]
    val s2 = b_sample[1]
    val dist = match_bob(s1, s2)
    val tmp50 = builder.yaoCircuit.putGTGate(min_dist, dist)
    val mux = builder.yaoCircuit.putMUXGate(dist, min_dist, tmp50)
    min_dist = mux
    i_2 += 1
}

val out = builder.yaoCircuit.putOUTGate(min_dist, Role.ALL)
executeABYCircuit(aby)
println(out.clearValue32.toInt())
}

else => throw ViaductInterpreterError('unknown host')
}

B.3 Interval

This benchmarks computes the interval in which Alice and Bob’s private points reside, and then checks whether Chuck’s private point resides in the interval. In the compiled implementation generated by the Viaduct compiler, Alice and Bob execute an MPC protocol to compute the interval in which their points lie (Lines 23–line 29). They then send the interval to Chuck, who sends either Alice or Bob a zero-knowledge proof to attest whether his point lies within the interval (line 44). If Alice receives the zero-knowledge proof, she verifies and then sends the result to Bob, and then they both output the result. The case where Bob receives the zero-knowledge proof is symmetric.
var min_point : int{A ∧ B} = points[0];
var max_point : int{A ∧ B} = points[0];

for (var i: int{A ∧ B ∧ C⁻} = 1; i < num_points; i += 1) {
  min_point = min(min_point, points[i]);
  max_point = max(max_point, points[i]);
}

val min_point_public : int{A ∧ B ∧ C⁻} =
  declassify min_point to {A ∧ B ∧ C⁻};
val max_point_public : int{A ∧ B ∧ C⁻} =
  declassify max_point to {A ∧ B ∧ C⁻};
val min_point_trusted : int{A ∧ B ∧ C} =
  endorse min_point_public from {A ∧ B ∧ C⁻};
val max_point_trusted : int{A ∧ B ∧ C} =
  endorse max_point_public from {A ∧ B ∧ C⁻};
val in_interval : bool{C ∧ (A ∧ B)⁻} =
  min_point_trusted <= chuck_point ∧ chuck_point <= max_point_trusted;
// Chuck doesn’t need to trust this because
// it will not be part of his output
val in_interval_public : bool{A ∧ B ∧ C⁻} =
  declassify in_interval to {A ∧ B ∧ C⁻};
output in_interval_public to alice;
output in_interval_public to bob;

B.4 k-means clustering

This benchmark runs a k-means clustering algorithm over Alice and Bob’s private data points. The compiled implementation executes the algorithm in an MPC protocol (Lines 25–79). After the algorithm finishes, the coordinates of the cluster centroids are declassified to both participants (Lines 82–86).

// Chuck doesn’t need to trust this because
// it will not be part of his output
output in_interval_public to alice;
output in_interval_public to bob;
for (var i: int{A ⊓ B} = 0; i < b_len * dim; i += 1) {
    data[(a_len*dim) + i] = input int from bob;
}

val clusters = Array[int]{A ⊓ B}(num_clusters * dim);

// initialize by picking data points as centroids in a stride
val stride: int{A ⊓ B} = len / num_clusters;
for (var c: int{A ⊓ B} = 0; c < num_clusters; c += 1) {
    for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
        clusters[(c*dim)+d] = data[(stride*c*dim)+d];
    }
}

for (var iter: int{A ⊓ B} = 0; iter < num_iter; iter += 1) {
    // assign points to clusters
    val best_clusters = Array[int]{A ⊓ B}(len);
    for (var i: int = 0; i < len; i += 1) {
        // initialize to first cluster
        var best_dist : int{A ⊓ B} = 0;
        var best_cluster : int{A ⊓ B} = 0;
        for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
            val sub : int{A ⊓ B} = data[(i*dim)+d] - clusters[d];
            best_dist += sub * sub;
        }
        for (var c: int{A ⊓ B} = 1; c < num_clusters; c += 1) {
            var dist : int{A ⊓ B} = 0;
            for (var d: int{A ⊓ B}; d < dim; d += 1) {
                val sub : int{A ⊓ B} = data[(i*dim)+d] - clusters[(c*dim)+d];
                dist += sub * sub;
            }
            best_cluster = dist < best_dist ? c : best_cluster;
        }
        best_clusters[i] = best_cluster;
    }

    // update cluster centroids
    for (var c: int{A ⊓ B} = 0; c < num_clusters; c += 1) {
        val new_centroid_sum = Array[int]{A ⊓ B}(dim);
        var num_points: int{A ⊓ B} = 0;
        for (var i: int = 0; i < len; i += 1) {
            val in_cluster : bool{A ⊓ B} = best_clusters[i] == c;
            for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
                new_centroid_sum[d] += in_cluster ? data[(i*dim)+d] : 0;
            }
            if (in_cluster) {
                num_points += 1;
            }
        }
    }
}
for (var d: int[A ∩ B] = 0; d < dim; d += 1) {
    clusters[(c*dim)+d] = num_points > 0 ?
        (new_centroid_sum[d] / num_points) : clusters[(c*dim)+d]);
}

// declassify clusters
for (var h: int[A ∩ B] = 0; h < num_clusters * dim; h += 1) {
    val public_cluster: int[A ∩ B] = declassify clusters[h] to (A ∩ B);
    output public_cluster to alice;
    output public_cluster to bob;
}

The program is compiled to one semantically equivalent to the Kotlin program below that uses ABY directly.

fun kmeans(host: Host, aby: ABYParty, builder: ABYCircuitBuilder) {
    val a_len = 50
    val b_len = 50
    val len = a_len + b_len
    val dim = 2
    val num_clusters = 4
    val num_iterations = 3

    // YaoABY
    val data = Array<Share?>((len * dim) { null }

    when (host) {
        'alice' => {
            var i = 0
            while (i < a_len * dim) {
                val x = input.nextInt()
                data[i] = builder.yaoCircuit.putINGate(x.toBigInteger(), BITLEN, builder.role)
                i += 1
            }
            var i_1 = 0
            while (i_1 < b_len * dim) {
                data[(a_len * dim) + i_1] = builder.yaoCircuit.putDummyINGate(BITLEN)
                i_1 += 1
            }
        }
        'bob' => {
            var i = 0
            while (i < a_len * dim) {
                data[i] = builder.yaoCircuit.putDummyINGate(BITLEN)
                i += 1
            }
            var i_1 = 0
            while (i_1 < b_len * dim) {

```
val x = input.nextInt()
data[(a_len * dim) + i_1] =
    builder.yaoCircuit.putINGate(x.toBigInteger(), BITLEN, builder.role)
i_1 += 1
}
}
else => throw Error('unknown host')
}

// ArithABY
val clusters = Array<Share?><(num_clusters * dim) { null }
val stride = len / num_clusters

var c = 0
while (c < num_clusters) {
    var d = 0
    while (d < dim) {
        clusters[(c * dim) + d] =
            builder.arithCircuit.putY2AGate(data[(stride * c * dim) + d], builder.boolCircuit)
        d += 1
    }
    c += 1
}

var iter = 0
while (iter < num_iterations) {
    // YaoABY
    val best_clusters = Array<Share?><(len) { null }

    // assignment phase
    var i = 0
    while (i < len) {
        var best_dist = builder.arithCircuit.putCONSGate(0.toBigInteger(), BITLEN)
        var best_cluster = builder.yaoCircuit.putCONSGate(0.toBigInteger(), BITLEN)

        // initialize point to first cluster
        var d = 0
        while (d < dim) {
            val tmp62 =
                builder.arithCircuit.putB2AGate(
                    builder.boolCircuit.putY2BGate(data[(i * dim) + d])
                )
            val sub = builder.arithCircuit.putSUBGate(tmp62, clusters[d])
            val tmp68 = builder.arithCircuit.putMULGate(sub, sub)
            best_dist = builder.arithCircuit.putADDGate(best_dist, tmp68)
            d += 1
        }

        // assign point to nearest cluster
        var c2 = 1
        while (c2 < num_clusters) {
            var dist = builder.arithCircuit.putCONSGate(0.toBigInteger(), BITLEN)
var d2 = 0
while (d2 < dim) {
    val tmp80 =
        builder.arithCircuit.putB2AGate(
            builder.boolCircuit.putY2BGate(data[(i * dim) + d2])
        )
    val sub = builder.arithCircuit.putSUBGate(tmp80, clusters[(c2 * dim) + d2])
    val tmp90 = builder.arithCircuit.putMULGate(sub, sub)
dist = builder.arithCircuit.putADDGate(dist, tmp90)
d2 += 1
}

val tmp91 = builder.yaoCircuit.putA2YGate(dist)
val tmp92 = builder.yaoCircuit.putA2YGate(best_dist)
val tmp93 = builder.yaoCircuit.putGTGate(tmp92, tmp91)
val tmp94 = builder.yaoCircuit.putCONSGate(c2.toBigInteger(), BITLEN)
val tmp96 = builder.yaoCircuit.putMUXGate(tmp94, best_cluster, tmp93)
best_cluster = tmp96
c2 += 1
}

best_clusters[i] = best_cluster
i += 1
}

// update phase
var c3 = 0
while (c3 < num_clusters) {
    // YaoABY
    val new_centroid_sum = Array<Share<?>>(dim) {
        builder.yaoCircuit.putCONSGate(0.toBigInteger(), BITLEN)
    }
    var num_points = builder.yaoCircuit.putCONSGate(0.toBigInteger(), BITLEN)
    var i2 = 0
    while (i2 < len) {
        val tmp108 = builder.yaoCircuit.putCONSGate(c3.toBigInteger(), BITLEN)
        val in_cluster = builder.yaoCircuit.putEQGate(best_clusters[i2], tmp108)
        var d3 = 0
        while (d3 < dim) {
            val tmp121 =
                builder.yaoCircuit.putMUXGate(
                    data[(i2 * dim) + d3],
                    builder.yaoCircuit.putCONSGate(0.toBigInteger(), BITLEN),
                    in_cluster
                )
            new_centroid_sum[d3] = builder.yaoCircuit.putADDGate(new_centroid_sum[d3], tmp121)
d3 += 1
        }
        val op =
            builder.yaoCircuit.putADDGate(
                num_points,
                builder.yaoCircuit.putCONSGate(1.toBigInteger(), BITLEN)
            )

    }
val mux = builder.yaoCircuit.putMUXGate(op, num_points, in_cluster)
    num_points = mux
    i2 += 1
}

var d4 = 0
while (d4 < dim) {
    val tmp132 =
        builder.yaoCircuit.putGTGate(
            num_points,
            builder.yaoCircuit.putCONSGate(0.toBigInteger(), BITLEN)
        )

    val tmp136 =
        Aby.putInt32DIVGate(builder.yaoCircuit, num_points, new_centroid_sum[d4])

    val tmp142 =
        builder.yaoCircuit.putA2YGate(clusters[(c3 * dim) + d4])

    clusters[(c3 * dim) + d4] =
        builder.arithCircuit.putB2AGate(
            builder.boolCircuit.putY2BGate(
                builder.yaoCircuit.putMUXGate(tmp136, tmp142, tmp132)
            )
        )

    d4 += 1
}

    c3 += 1
}

iter += 1
}

var h = 0
var out_gates = Array<Share?><(num_clusters * dim) {
    builder.arithCircuit.putCONSGate(0.toBigInteger(), BITLEN)
}
while (h < num_clusters * dim) {
    out_gates[h] = builder.arithCircuit.putOUTGate(clusters[h], Role.ALL)
    h += 1
}

aby.execCircuit()

var i = 0
while (i < num_clusters * dim) {
    println(out_gates[i]!!.clearValue32.toInt())
    i += 1
}
}
B.5 Rock–Paper–Scissors

Alice and Bob play a game of rock–paper–scissors. In the compiled implementation, Alice and Bob input their moves ahead of time and send each other commitments to their moves (Lines 10–13). Then the turns of the game are played by opening the commitments to Alice and Bob’s moves for that turn and awarding the winning player a point (Lines 19–53). If a player’s input is invalid, the other player is awarded a point. At the end of the game, the winner is determined and sent as output to the players (Lines 56–58).

```plaintext
1 host alice : {A}
2 host bob : {B}
3
4 val num_turns: int{A ∩ B} = 3;
5 var a_score: int{A ∩ B} = 0;
6 var b_score: int{A ∩ B} = 0;
7 val a_moves = Array[int]{A ∩ B}(num_turns);
8 val b_moves = Array[int]{B ∩ A}(num_turns);
9
10 for (var i: int{A ∩ B} = 0; i < num_turns; i += 1) {
11   a_moves[i] = endorse (input int from alice) from {A};
12   b_moves[i] = endorse (input int from bob) from {B};
13 }
14
15 for (var turn: int{A ∩ B} = 0; turn < num_turns; turn += 1) {
16   val a_move: int{A ∩ B} = a_moves[turn];
17   val b_move: int{B ∩ A} = b_moves[turn];
18
19   val a_move_public: int{A ∩ B} = declassify a_move to {A ∩ B};
20   val b_move_public: int{A ∩ B} = declassify b_move to {A ∩ B};
21
22   // 1 = rock; 2 = paper; 3 = scissors;
23   val a_valid: bool{A ∩ B} = 1 <= a_move_public ∧ a_move_public <= 3;
24   val b_valid: bool{A ∩ B} = 1 <= b_move_public ∧ b_move_public <= 3;
25
26   // alice cheats
27   if (!a_valid ∧ b_valid) {
28     b_score += 1;
29   }
30
31   // bob cheats
32   if (a_valid ∧ !b_valid) {
33     a_score += 1;
34   }
35
36   // neither cheat
37   if (a_valid ∧ b_valid) {
38     if (a_move_public < b_move_public ∧ b_move_public < 3) {
39       b_score += 1;
40     }
41     if (b_move_public < a_move_public ∧ a_move_public < 3) {
42       a_score += 1;
43     }
44     if (a_move_public == 1 ∧ b_move_public == 3) {
45       a_score += 1;
46   }
47 ```
Viaduct

```java
if (b_move_public == 1 ∧ a_move_public == 3) {
    b_score += 1;
}
```
output winner to bob;

The program is compiled to one semantically equivalent to the Kotlin program below that uses ABY directly.

```kotlin
fun twoRoundBidding(host: Host, aby: ABYParty, builder: ABYCircuitBuilder) {
    val n = 500
    when (host) {
        'alice' => {
            val abids1 = Array<Int>(n) { 0 }
            val abids2 = Array<Int>(n) { 0 }

            var i = 0
            while (i < n) {
                abids1[i] = input.nextInt()
                i += 1
            }

            var i_1 = 0
            while (i_1 < n) {
                val tmp15 =
                    builder.yaoCircuit.putINGate(
                        abids1[i_1].toBigInteger(), BITLEN, builder.role
                    )
                val tmp17 = builder.yaoCircuit.putDummyINGate(BITLEN)
                val tmp18 = builder.yaoCircuit.putGTGate(tmp17, tmp15)
                val tmp19 = builder.yaoCircuit.putOUTGate(tmp18, Role.ALL)
                aby.execCircuit()
                val winner = tmp19.clearValue32.toInt()
                aby.reset()
                println(winner)
                i_1 += 1
            }

            var i_2 = 0
            while (i_2 < n) {
                abids1[i_2] = input.nextInt()
                i_2 += 1
            }

            var i_3 = 0
            while (i_3 < n) {
                val abid =
                    builder.yaoCircuit.putINGate(
                        ((abids1[i_3] + abids2[i_3]) / 2).toBigInteger(), BITLEN,
                        builder.role
                    )
                val bbid = builder.yaoCircuit.putDummyINGate(BITLEN)
                val tmp46 = builder.yaoCircuit.putGTGate(bbid, abid)
```
val tmp47 = builder.yaoCircuit.putOUTGate(tmp46, Role.ALL)
aby.execCircuit()
val winner_1 = tmp47.clearValue32.toInt()
aby.reset()
println(winner_1)
i_3 += 1
}

'bob' => {
    val bbids1 = Array<Int>(n) { 0 }
    val bbids2 = Array<Int>(n) { 0 }
    var i = 0
    while (i < n) {
        bbids1[i] = input.nextInt()
        i += 1
    }
    var i_1 = 0
    while (i_1 < n) {
        val tmp15 = builder.yaoCircuit.putDummyINGate(BITLEN)
        val tmp17 = builder.yaoCircuit.putINGate(
            bbids1[i_1].toBigInteger(), BITLEN, builder.role
        )
        val tmp18 = builder.yaoCircuit.putGTGate(tmp17, tmp15)
        val tmp19 = builder.yaoCircuit.putOUTGate(tmp18, Role.ALL)
        aby.execCircuit()
        val winner = tmp19.clearValue32.toInt()
        aby.reset()
        println(winner)
        i_1 += 1
    }
    var i_2 = 0
    while (i_2 < n) {
        bbids1[i_2] = input.nextInt()
        i_2 += 1
    }
    var i_3 = 0
    while (i_3 < n) {
        val abid = builder.yaoCircuit.putDummyINGate(BITLEN)
val bbid =
    builder.yaoCircuit.putINGate((
        bbids1[i_3] + bbids2[i_3]) / 2).toBigInteger(),
    BITLEN,
    builder.role
)
val tmp46 = builder.yaoCircuit.putGTGate(bbid, abid)
val tmp47 = builder.yaoCircuit.putOUTGate(tmp46, Role.ALL)
aby.execCircuit()
val winner_1 = tmp47.clearValue32.toInt()
aby.reset()
println(winner_1)
i_3 += 1
else => throw ViaductInterpreterError('unknown host')
}