A Trusted Third-Party Computation Service

Sameer Ajmani Robert Morris Barbara Liskov

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139, USA
{ajmani,rtm,liskov}@lcs.mit.edu

Abstract

We present TEP, a system that supports general-purpose shared computation between mutually-distrusting parties. TEP is useful for applications, such as auctions and tax preparation, that use private information from multiple participants. Such applications cannot be run on any one participant's computer without sacrificing the other participants' privacy. TEP acts as a trusted service that hosts the sensitive parts of such applications.

TEP uses a Java VM to load and run computations on behalf of clients. TEP uses Java security mechanisms and cryptographic protocols to ensure that (1) a program can communicate only with the specific participants identified for a computation and (2) each participant knows exactly what program is being run and who the other participants are. This lets participants determine whether information they send to the computation can be exposed to other participants; we show how static analysis greatly simplifies this task. Example programs show that the TEP model is useful and easy to program; benchmarks show that the TEP prototype implementation is fast enough to be practical.

Keywords: multiparty computation, trusted third party, information flow, privacy

This research was supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory.

1 Introduction

Sharing private data in a computation presents a paradox. How can two parties combine their private data in a computation without revealing their data to one another? Many systems, such as online auctions, solve this problem by introducing a trusted third party to run the computation. However, such systems are usually specialized to a single application and provide only vague guarantees on the system's ability to control information leaks. This paper presents the Trusted Execution Platform (TEP), a new system that supports general-purpose multiparty computation with specific guarantees on information leaks.

Consider an online tax-preparation application called "WebTax". WebTax uses a proprietary database provided by a tax preparer and requires private income data from the client. If WebTax runs on a computer controlled by either the client or the preparer, the privacy of the other party's data is compromised. Running WebTax on TEP protects the privacy of both parties.

There are two general types of difficulties that might afflict a trusted third-party computation service. First, the third party might turn out not to be trustworthy, due to either malice or incompetence. We have nothing new to offer here. We assume that TEP is trustworthy and simply guarantee that users of TEP can authenticate their connections to TEP.

The second type of difficulty arises when the participants in a shared computation try to trick each other. A participant might not realize that a shared computation's algorithms reveal its data to another participant. A participant might be misled about what computation it is participating in; for example, it might submit a bid to the wrong auction. A participant might understand the computation involved, but be mistaken about who else is participating. A participant might understand the computation and know the participants, but be misled about the role of each participant. Any of these errors could result in undesired exposure of private data, even if TEP is completely trustworthy.

The key to avoiding these errors is thorough identification. TEP provides enough information to each participant to allow a precise understanding of whom the participant's private data may be exposed to. TEP identifies specific properties about each computation to every participant involved in that computation:

TEP authentication of TEP itself

Program the exact bytecodes of the program used for a computation

Participants the participants in a computation

Roles the role each participant plays in the computation

Instance distinguishes between computations with the same program, participants, and roles.

To make these properties meaningful, TEP must guarantee that data used in a computation cannot escape through unidentified channels:

Isolation Ensure that data given to a program cannot leak to parties that are not participants.

Although TEP can prevent data from being exposed to non-participants, it is still up to each user to determine whether a program leaks his or her information to legitimate participants. Doing this requires inspection of the code. This inspection process is likely to be tedious and error prone, so we explain how inspection can be automated using static information-flow analysis.

Section 2 presents a design for TEP that provides the required identification and isolation. Section 3 details how the design is implemented. Section 4 describes how TEP incorporates static analysis, and Section 5 presents performance results for a prototype implementation. Finally, Section 6 discusses related work, and Section 7 concludes.

2 Design

This section describes how TEP satisfies its identification and isolation requirements. The design uses a "call back" model, in which a computation is invoked with a set of parameters that identify the participants. The computation initiates connections to the participants specified in the parameters. Participants run trusted local agents that accept connections from computations on TEP and handle the necessary authentication and policy decisions on their participant's behalf.

An alternate model is to allow participants to initiate connections to computations on TEP. This has the convenience of following the standard client-server model and allowing participants to join long-running computations dynamically. However, this adds complexity to TEP, since TEP must forward incoming connections to the correct computations. We plan to support this model in future work.

All communication between a computation running on TEP and the participants in the computation occurs through *channels*. A channel is a bidirectional communication connection between a program on TEP and a participant. A program on TEP can only communicate with participants using channels, which allows TEP to authenticate each source and sink for the data used in the computation.

2.1 TEP Identification

Whenever TEP opens a channel to a participant, it carries out a two-way authentication protocol in which it identifies itself to the participant. This allows the participant to be certain that it is communicating with TEP. We assume that participants can find the public key for TEP from a key distribution authority.

2.2 Program Identification

To identify a program, TEP creates a fingerprint of the program bytecodes. The fingerprint is calculated using a cryptographic hash of the bytecodes in a canonical order. Given this fingerprint, a participant can determine (with high probability) what program is running on TEP and so base its decision about whether to participate on local knowledge about that program.

2.3 Participant Identification

Each participant is identified by a public key, which it uses to authenticate itself when TEP opens a channel to it. TEP checks that this key matches the one for the expected participant for that channel.

2.4 Role Identification

In any computation, a participant plays a particular *role*. A role is a name for the part a participant plays in a computation, such as *client* or *bidder*. Each channel specifies a role for the participant it connects to; a participant can refuse the connection if the role isn't what it expects. Additionally, roles help users understand how their data is used in the computation, since there will typically be a direct connection between a role and a program variable.

2.5 Instance Identification

It is possible that multiple computations with the same program, participants, and roles are running on TEP simultaneously. To distinguish these instances, TEP assigns each computation an *instance identifier*. This identifier is a *nonce*: a new value that, with high probability, is different from the identifier for any other computation with the same program, participants, and roles. This identifier can be used to identify the computation and its results after the fact, e.g., in an auditing procedure.

2.6 Isolation

TEP must ensure that data used in a computation does not leak to any parties outside that computation. This requires that TEP restrict a program's behavior in a few ways. First, TEP restricts all program input and output to channels, so that TEP can intervene before any data enters or leaves the system. Furthermore, all communication over these channels is encrypted and authenticated to prevent eavesdropping and impersonation.

Next, since TEP is a multi-process system, TEP must isolate the data areas and name spaces for concurrent programs from one another.

TEP must also ensure that programs cannot corrupt TEP's own process or access system resources directly, since this might allow programs to circumvent TEP's protection mechanisms. TEP denies programs direct access to the network, disk, and runtime system, since access to any of these could allow programs to leak data or attack on TEP directly. This means programs cannot store any persistent state on TEP between invocations. This is usually acceptable, since programs can send intermediate results to participants over channels and read them back later.

3 Implementation

This section describes how TEP's design can be implemented efficiently.

Programs for TEP are written in Java. Java provides many features that make the implementation straightforward, such as type safety (backed up by bytecode verification) and the ability to load code dynamically [7].

An example of a Java program for TEP is given in Figure 1. WebTax is a tax preparation application with two participants: a client that provides private income data and a tax preparer that provides a proprietary tax database.

WebTax reads data from both participants, calculates the client's tax, and writes the result back to the client. This section will refer to this example as each new concept is introduced.

3.1 Code Structure

Any program that can be run by TEP must provide exactly one `run` method. The `run` method must be public and static, return `void`, and throw no exceptions. This ensures that TEP can invoke the `run` method and that TEP need not provide any specific information on to the participants how the the method terminates.

Additionally, the parameters of the `run` method are limited to the type `Participant`; we discuss this type further below. For example, WebTax takes two participants as arguments, the client and the tax preparer.

3.2 Invocation

Anyone can invoke a computation on TEP by specifying a URL for the program, a fingerprint for the bytecodes, and a set of participant bindings. Say a client with key `Kc` wants to invoke WebTax for a tax preparer with key `Kp`. The client must provide TEP with a URL for WebTax, a fingerprint for the code, and a role, host, port, and key for itself and the tax preparer (presented here in an XML format):

```
<Invocation url="http://tax.com/app.jar"
    md5-hash="L1vqv6VrqBk5OVCXTdhOUA==">
  <Participant role="client"
               host="host-c.domain.com"
               port=1234
               key=Kc />
  <Participant role="preparer"
               host="server.tax.com"
               port=6060
               key=Kp />
</Invocation>
```

Typically, the user does not need to provide this information directly. For example, the tax preparer might have a web site that would allow a client to sign up for tax preparation by filling out a form. Then the form would be processed automatically to provide the invocation description and send it to TEP.

When TEP receives an invocation request, it fetches the WebTax application from the URL and checks that it matches its fingerprint. Then it runs the bytecode verifier on every class in the entire application. If either of these fails, TEP refuses the request.

TEP then looks for WebTax's `run` method. The number of participants in `run`'s signature must match the number of participants specified by the invoker. Given the above computation request, WebTax's `run` method must have this signature:

```
public static void run(Participant,
                       Participant);
```

the invocation request is incompatible with the method signature, TEP refuses the request.

TEP next creates objects that represent the participants: one for each participant in the request. TEP then runs the application, passing it the `Participant` objects in the order specified in the invocation request.

The participant objects are of type `Participant`, which is a Java class that is provided by TEP for use in programs that will run on TEP. Each `Participant` contains a role, host, port, and key:

```
public class Participant {
  final public String role;
  final public String host;
  final public int port;
  final public PublicKey key;
  Participant(String role,
              String host,
              int port,
              PublicKey key);
}
```

Participant is an immutable type that has a *package visible* constructor. This means that only TEP's code can create `Participant` objects. Programs on TEP use participants for channel creation, as explained in the next section.

To support variable numbers of participants, TEP provides the ability to create an array of participants that share the same role. For example, say participants with keys `Ka` and `Kb` wish to bid in an auction. Each participant can fill out a form to provide an auction site with their host, port, and key information (this is not a

```
public class WebTax
{
  public static void run(Participant clnt, Participant prep)
  {
    try {
      // read client's income data
      Channel clientChan = new Channel(clnt);
      Income income = new Income(clientChan.getInputStream());

      //get tax preparer's channel
      Channel prepChan = new Channel(prep);

      // calculate taxes
      Tax tax = calcTax(income, prepChan);

      // send tax to the client
      tax.writeTo(clientChan.getOutputStream());
    } catch (Exception e) {
      // ignored
    }
  }
}
```

Figure 1: WebTax Java Code

serious privacy risk since the agents running at the participants can filter out any unsolicited computations). The auction site then invokes the auction on TEP with the command:

```
<Invocation url="http://tbay.com/app.jar"
      md5-hash="6Pjqrrxv6yi+wOYDMdpROA=="
      app-ident="1997 Ford Explorer">
  <Participant role="auctioneer"
               host="tbay.com"
               port=7070
               key=Kt />
  <ParticipantArray role="bidder">
    <Participant host="host-a.domain.com"
                 port=1234
                 key=Ka />
    <Participant host="host-b.domain.com"
                 port=4321
                 key=Kb />
  </ParticipantArray>
</Invocation>
```

In an auction, many participants fill the same role (bidder). This is represented in the request as a list of participants nested under a `ParticipantArray` element that specifies the role. The corresponding `run` method for this request is:

```
public static void run(Participant,
                       Participant[]);
```

Another point about this example is that the invocation description includes an *application identifier*. The `app-ident` field allows the invoker to provide an uninterpreted string that can be used in an application-specific way. In this case, it is being used to identify the particular auction being run. While this field isn't strictly necessary, it can be useful to allow an application to easily distinguish different computations that use the same program. The application identifier field is separate from the instance identifier assigned by TEP.

3.3 Channels

TEP provides programmers with an API for creating and using channels. The implementation of this API is stored on TEP and linked with a program when it is loaded. The API defines one class, `Channel`:

```
public class Channel {
  public Channel(Participant p)
    throws AccessControlException,
```

```
            GeneralSecurityException,
            IOException;
  public Participant getParticipant();
  public InputStream getInputStream();
  public OutputStream getOutputStream();
}
```

`Channel` contains four instance methods: a constructor and three accessors. A program creates a channel by calling the constructor with a `Participant` as an argument. The constructor opens a secure connection to the given participant and throws an exception if an error occurs. `getParticipant` provides access to the participant used to create the channel. The other accessors provide access to the channel using the standard Java input and output stream abstractions [8]. TEP's underlying implementation of these streams provides secure information transfer.

TEP creates a channel as follows:

1. connects to `p.host` and `p.port`, where `p` is the `Participant` passed to the channel (throws an `IOException` if this fails)

2. authenticates the channel and checks that the peer's public key matches `p.key` (throws an `AccessControlException` if this fails)

3. uses encryption and message authentication codes (MACs) to protect the channel's privacy and integrity (throws a `GeneralSecurityException` if this fails)

4. sends the computation information (the program fingerprint, the application identifier, this channel's participant, the other participants, and the instance identifier) to the peer in a format similar to the one for invoking a computation

The required authentication and encryption can be implemented using any secure session protocol, such as SSL [21].

The peer checks that it is connected to the correct program and that its role is what it expects; it disconnects if not. The peer may also check the application identifier and the other participants; for example, the client of WebTax may check to be sure that the other participant is the tax preparer it expects. If TEP receives no response from the participant, it times out and throws an `IOException` to the program.

3.4 Example: WebTax

The Java code for WebTax is presented in Figure 1. WebTax defines a `run` method, which is the top-level method invoked by TEP. When TEP receives a computation request for WebTax, it creates two `Participant` objects, one for the client and one for the tax preparer.

WebTax creates a channel to the client using the first participant. If the channel is created successfully (i.e., the agent acting as the client accepts the channel), WebTax reads the client's income data by constructing an `Income` object from the channel's input stream. `Income` is a class included with WebTax that knows how to read its value from a Java `InputStream`.

Next WebTax creates a channel to the tax preparer and calls the method `calcTax` to calculate the client's tax. This method uses the preparer's channel to read the preparer's database and uses that information along with the client's income data to compute the taxes. Finally, WebTax writes the result to the client's output stream by calling the result's `writeTo` method on the client's output stream.

This program leaks data. Since the client's `tax` incorporates information read from the preparer's database, some amount of the preparer's private information is leaked to the client. This is presumably acceptable for the tax preparer (or the program would not have been made available for general use), and without this leak no useful computation can be shared. Section 4 will show how static analysis can detect such leaks.

3.5 Example: Auction

The Auction example in Figure 2 implements a silent auction for a set of bidders. This application is invoked with two types of participants: a single auctioneer and a set of bidders. The auctioneer invokes the auction and identifies the item up for bid by setting an application identifier. The auctioneer also sets the reserve price. The bidders each check the application identifier to confirm they are in the correct auction, send in a bid, then read the winning bid at the end of the auction.

First, the program connects to the auctioneer. If this fails, the program exits, since there is no reason to continue (it could also contact each bidder to report the error). The program then uses the channel to read the reserve price.

Next, the program creates a channel for each bidder. If the bidder discovers that it is in the wrong auction, it refuses the connection. In this case, the program will get an `IOException` and continue to the next bidder. After all the bids have been read and the winning bid has been calculated, the program reports the winning bid to each bidder.

3.6 Policies

When a computation connects to a participant, TEP provides the participant with enough information to determine whether that computation can leak the participant's data. Each participant decides independently whether to participate in a computation according to its own local security policies.

Since policies are local, participants can automate policy processing using a *policy engine* [23]. A policy engine can answer queries about policy and, given a computation, decide whether the participant should participate. These decisions might consider whether the computation can leak the participant's data and, if so, whether the participant considers the leak acceptable.

By automating the approval process, policy engines enable participants to connect their own programs to TEP over channels. Typically, a particular shared computation will require a specialized client-side agent to interact with the computation, and that agent acts as the policy engine for that computation. For example, an agent for WebTax accepts a connection from a computation on TEP and checks that the computation has the correct parameters.

A client-side agent might help a participant "sign up" for a group computation, such as an auction, and then wait for a connection from TEP. When TEP connects to the agent, the agent checks that the computation is using the correct bytecodes and that it is bound to the correct channel. The agent might also check the bindings for the other participants in the computation.

A participant must trust its agent, since the agent has access to the participant's local data. This is a serious risk, since an agent could reveal the participant's data directly to other parties on the network. To alleviate this risk, TEP provides the `TepAgent` class:

```
public class TepAgent {
  public TepAgent(KeyPair localKeys);
  public Invocation getInvocation();
  public InputStream getInputStream();
  public OutputStream getOutputStream();
}
```

When constructed, `TepAgent` waits for a connection from a computation on TEP, authenticates TEP, authenticates itself to TEP using the local participant's keys, and reads the invocation information from the connection. `TepAgent` provides accessors for the invocation information and the connection to the computation. `Invocation` is a Java type that provides the same information used to invoke the computation.

Application-specific agents can be built on top of `TepAgent`. Participants can then deny application agents direct access to the network (using Java-style security policies [10]), since those agents should only communicate to TEP using `TepAgent`. `TepAgent` can provide additional protection mechanisms, such as requesting direct participant approval of a computation. Since there can be more than one TEP server, each server should provide a signed implementation of `TepAgent` (or an appropriate certificate) that will properly authenticate that server.

3.7 Isolation

TEP must isolate programs from each other to ensure they cannot share data outside their

```
public class Auction
{
  public static void run(Participant a, Participant bs[])
  {
    try {
      // create a channel to the auctioneer and
      // read the reserve price
      Channel auctionChan = new Channel("auctioneer", a);
      Bid reserve = new Bid(auctionChan.getInputStream());
      Bid max = reserve; // the current max bid
    } catch (Exception e) {
      return; // bad auctioneer: unrecoverable
    }

    // create an array of channels for the bidders
    Channel bidChans[] = new Channel[bs.length];

    // open a channel to each bidder and read each bid,
    // updating max as needed
    for (int i = 0; i < bs.length; i++) {
      try {
        bidChans[i] = new Channel("bidder", bs[i]);
        Bid bid = new Bid(bidChans[i].getInputStream());
        if (bid.value > max.value) max = bid;
      } catch (Exception e) {
        continue; // bad bidder: move on to the next one
      }
    }

    // send the winning bid to each bidder
    for (int i = 0; i < bs.length; i++) {
      try {
        max.writeTo(bidChans[i].getOutputStream());
      } catch (IOException e) { /* ignored */ }
    }
  }
}
```

Figure 2: Auction Java Code

channels. TEP must also prevent programs from accessing system resources directly, since otherwise programs could use the disk or the network to leak data. TEP implements these measures using specific features of Java: type safety, classloaders, and security managers.

TEP must ensure the memory safety of each program. A number of techniques have been developed to isolate dynamically-loaded code [22, 18, 16]. TEP depends on Java's type safety to provide this protection.

TEP runs each program in its own thread in TEP's Java virtual machine (JVM). TEP creates a new instance of a custom Java classloader for each program. This classloader provides each program with its own class namespace and static data area [11]. The classloader only allows a program to load the TEP-provided classes `Channel` and `Participant` and the application classes that are included in the program's JAR file [9]. Java classloaders provide a form of *name space management* that prevents programs from accessing each other's code and data [23]. This allows multiple copies of the same program to execute independently in the same JVM.

TEP uses Java's security manager to deny programs direct access to system resources, such as the Java runtime environment, the disk, and the network [10]. This prevents programs from circumventing TEP's channel API and ensures that TEP can intervene before any data leaves the system. This also protects TEP's own code from attack by the programs it runs. Although these mechanisms suffice, other research has explored providing much stronger process isolation in Java [3].

TEP does not use any resource allocation scheme to protect programs from CPU or network starvation. Although unnecessary for privacy, such a scheme is needed to protect TEP and the programs it runs against denial-of-service attacks. Adding resource control would require that programs include information about their resource needs and would require that TEP either monitor programs' resource consumption [18] or check them statically [16].

Finally, TEP makes no claims on covert channels, such as timing channels. However, static information-flow analysis will allow TEP to detect implicit information flows in the control structure of a program, as discussed in Section 4.

3.8 Extensions

3.8.1 Auditing

Even though TEP identifies each computation to the participants, participants might disagree about the result of a computation (for example, the winner of an auction or the item up for bid). The solution to this problem is *auditing*: keeping logs of computations for use in settling future disputes.

TEP can act as a trusted auditor by recording all data sent to a program over each channel. Provided programs are deterministic, TEP can "replay" a recorded program by simulating the channels. To ensure deterministic behavior, TEP must deny program access to all sources of nondeterminism, such as the system clock. This is not a serious limitation since such information can be provided by participants over channels (and so logged by TEP). To match audit logs with particular computations, participants need only provide the instance identifier assigned by TEP for that computation.

Keeping audit logs places a heavy load on TEP and introduces the risk that TEP might leak the logs, since they must be stored for an extended period of time. A solution is for TEP to encrypt a computation's logs once the computation is complete and store the encrypted logs in a reliable offline database. The computation information must be public for indexing, but the privacy of the logs themselves is protected by encryption.

3.8.2 Using a PKI for Role Bindings

In TEP, each computation has its own bindings from participant's roles to keys. This relation maps naturally to name-key bindings in a public key infrastructure (PKI), such as PKIX (X.509) [17] or SPKI [19]. Instead of inspecting the role bindings for a computation at runtime, participants can create role bindings *before* a computation by issuing the appropriate certificates.

Using a PKI, an invocation no longer needs to include keys in the role bindings. Instead, an invocation needs to identify the root naming authority. This is the public key of the certification authority (CA) that issues the role-key bindings for the computation.

When a computation opens a channel, TEP verifies that the peer's key is bound to the channel's role by certificates issued by the root CA. TEP sends the root CA key to each participant along with the other computation information, so each participant can verify that TEP is using the correct bindings.

This structure has the benefit of simplifying participants' computation approval process at runtime, since each participant need only check the root CA key [1]. A drawback is that TEP must now verify role-key bindings when channels are created. If this involves searching a PKI, this can be very slow. Another issue is that a PKI is less flexible than including role bindings with each computation, since certificate propagation and revocation can take time. This is a drawback when bindings need to change often.

A solution to both these problems is to include the required certificates with the invocation, as in proof-carrying authentication [2]. Role bindings can be represented as chains of name certificates in SDSI/SPKI-style linked local name spaces; these chains must be provided to TEP when the computation with those bindings is invoked. This alternative still requires that TEP verify the proofs of each property; but this may be faster than requiring that each participant individually approve the bindings.

A client using a PKI can invoke WebTax with these parameters:

```
<Invocation url="http://tax.com/app.jar"
    md5-hash="L1vqv6VrqBk5OVCXTdhOUA=="
    root-key=Kr>
  <Participant role="client"
               host="host-c.domain.com"
               port=1234 />
  <Participant role="preparer"
               host="server.tax.com"
               port=6060 />
  <Proof role="client" key=Kc>
    <!-- sequence of certificates -->
  </Proof>
  <Proof role="preparer" key=Kp>
    <!-- sequence of certificates -->
  </Proof>
</Invocation>
```

The top-level element specifies the root CA key with the `root-key` attribute. The participant elements no longer specify keys, since the role-key bindings are specified by certificates in the PKI. This invocation include proofs for these bindings, so TEP doesn't need to search the PKI for them. TEP sends this information, except for the proofs, to each participant in the computation. Given just the root CA key, the participants can determine whether TEP is using the correct bindings for the computation.

3.8.3 Anonymous Participants

Not all participants may want to be identified to every other participant. In an auction, bidders' identities are irrelevant to the computation, and a bidder may want hide its identity for a given auction. TEP can support this with *anonymous participants*: the invoker of a program can mark certain participants as anonymous, which means their public key, host, and port will not be sent to the other participants. However, TEP will report to each participant the number of anonymous participants in each role. This allows a participant to refuse to participate in a computation where roles should not be anonymous, such as the tax preparer in WebTax.

4 Information Flow Analysis

As described so far, TEP allows computations to share data with participants without leaking their data. But there is a major problem: how does a participant decide whether the program being run leaks his or her data? The only way this can be done so far is for the participant to carefully inspect the code. Unfortunately, inspection is tedious and error-prone, especially as programs become more complex.

For example, a client of WebTax would need to examine the code to be sure that his or her income information is not leaked to the tax preparer. This would require reading the code of the `calcTax` method and all the code it calls. This may very well be a huge program, so reading it and understanding whether it leaks information is a formidable undertaking. Furthermore, the code may be proprietary: the tax preparer may not want to allow the code to be read.

The solution to this problem is *static information-flow analysis*. This analysis checks a program at compile-time for information leaks. Any information that a program leaks must be explicitly *declassified*; the participant whose data is leaked must explicitly authorize that leak. For example, the tax preparer must authorize WebTax. The client, however, knows that it does not authorize WebTax, and so does not need to inspect the code (which also protects the tax preparer's privacy interests for the code).

Static analysis tracks not only explicit information flows through direct computation, but also implicit flows through the control flow of a program. For example, the statement `if (secret) channel.write(true)` leaks the value of `secret`. Static analysis catches this leak and makes it possible to find leaks that would be extremely difficult to find by inspection.

TEP supports static information-flow analysis using Myers' Java information flow language, Jif [15, 14]. Jif extends Java with decentralized information-flow *labels*, a type of annotation that converts information flow analysis to an extended form of type-checking. A Jif compiler or bytecode verifier can efficiently analyze a labelled Jif program to determine whether it leaks any data without explicitly declassifying it, and will reject the program if it does [14, 12].

4.1 Labels

Jif labels are annotations on the type of a variable that represent the privacy policies for that variable. Labels contain policies owned by *principals*, which are named entities that embody some authority in a program. Jif defines an ordering on labels that prevents data with a given label from being written to a variable with a less restrictive label, and so prevents data from being leaked as it moves between variables in a program.

Jif allows the roles in a computation to be captured directly in labels. In particular, a participant's role will be captured by the label on the `Participant` object created for that user. For example, the client's participant in WebTax would have type `Participant{client:}`. This label means that the labeled data item is readable only by the client. Jif labels have a far richer structure than shown here, but TEP does not use anything more.

To illustrate the power of Jif, the Jif version of WebTax is presented in Figure 3. The first thing to note is the use of labels in the header of the `run` method to capture the roles of the two participants. This header indicates that there are two principals involved in running the program, one representing the client, and the other, the tax preparer.

The next thing to notice is the use of channels. In Jif, channels are parameterized by the principal of the participant that will be using that channel. Such a channel restricts the labels of data that can be read and written to the channel, namely to have the label `{principal:}`. Thus the channel to the client will produce data labeled `{client:}` and will also only accept data labeled `{client:}`. Similarly the channel produced for the tax preparer produces and accepts only data with the label `{preparer:}`. Note that this prevents the client's data from being written to the preparer's channel and vice versa.

The tax computation produces a result that has a compound label `{client:, preparer:}`, since the result involves data from both parties. This result cannot be written to the client's

```
public class WebTax authority (preparer)
{
  public static void run(Participant{client:} clnt, Participant{preparer:} prep)
    where authority (preparer)
  {
    try {
      // read client's income data
      Channel[client] clientChan = new Channel[client](clnt);
      Income{client:} income = new Income(clientChan.getInputStream());

      //read preparer's tax database
      Channel[preparer] prepChan = new Channel[preparer](prep);

      // calculate taxes
      Tax{client:, preparer:} tax = calcTax(income, prepChan);

      // send tax to the client
      declassify(tax,{client:}).writeTo(clientChan.getOutputStream());
    } catch (Exception{client:} e) {
      // ignored
    } catch (Exception{preparer:} e) {
      // ignored
    }
  }
}
```

Figure 3: WebTax Jif Code

channel since its label doesn't match what the channel requires.

To handle this case, Jif allows data to be declassified. However, a program can declassify data only if it runs with the authority of the principal in the label being removed. WebTax runs with the authority of the tax preparer, as indicated in the `authority` clause in its header. Therefore, it can declassify the result and return it to the client.

However, WebTax does not run with the authority of the client. This means that it *cannot* remove any of the client's labels from the data, and therefore it is unable to leak the client's information to the tax preparer. Furthermore, the client can determine this by simple inspection of the header of WebTax; it is not necessary to read the code.

For other programs, however, things aren't so simple. For example, the bidders in the auction need to authorize the program to run with their authority so that it can let all of them know about the winning bid. Even so, less reading of code may be required because the code can be organized so that all declassification happens in the top level code. For example, even if the client of WebTax gave the program some authority (e.g., to release the name and address of the user to the tax preparer), it would be unnecessary to examine the code of `calcTax` provided it did not run with the client's authority, which can be determined by examining its header.

4.2 Runtime Authority

Jif is far more flexible than described in this paper; discussion of many of Jif's features has been omitted for brevity. However, one feature of Jif that is particularly helpful to TEP is Jif's ability to represent and reason about principals whose authority is unknown at compile-time. This feature makes it possible to protect distinct participants in a TEP computation from accidentally sharing data when they have the same role, such as the bidders in an auction. Supporting these features requires some simple extensions to TEP, which are described here.

Jif represents unknown authorities as *runtime principals*, which are first-class values of type `principal`. Runtime principals can be used in labels and so can represent the role of a participant in a TEP computation. Participants whose labels contain runtime principals cannot be compared directly, and so cannot accidentally share data.

When the authority of the participant needs to be known (eg. to do a declassification), the program must check whether the participant's authority can be represented with a known static authority. Jif has two features that make this check possible: the ability to compare the authorities of static and runtime principals and the ability to determine the value of an unknown label.

Runtime principals can be compared to static principals using an `actsfor` test in Jif. `actsfor` checks whether one principal is authorized to act on behalf of another, in which case the authorized principal can use the other's authority. In an auction, each bidder is a distinct participant, but a static principal (such as `bidder`) must be authorized to declassify each bidder's bid. An auction can use an `actsfor` test to check whether this authorization exists at runtime: if so, the declassification is allowed; otherwise, the auction must ignore that bidder's bid. Although this may seem similar to the original auction model, this model limits the use of `bidder`'s authority to only the code where it is needed. Jif supports a similar runtime test for labels that provides similar benefits.

To support Jif's ability to represent and reason about runtime authority, TEP must be extended in two ways. First, TEP must provide a way to represent principals at runtime. TEP does this by providing a class that stores the information about a runtime principal, such as the underlying participant's public key. Second, TEP must provide a way to compare principals. TEP does this using a *principal hierarchy*, which is a data structure that represents the `actsfor` relation between principals. TEP can then determine whether one principal can act for another by searching for a path between those principals in the principal hierarchy.

5 Prototype

5.1 Model

The prototype implementation of TEP is based on a design that uses a PKI, as described in Section 3.8.2. All policy decisions are made by TEP, rather than distributed to the participants [1]. The prototype both accepts certificate chains for policies [2] and supports on-demand policy generation using *certificate-chain discovery* [4, 1]. The prototype also automatically includes the invoker of a computation as one of the computation's participants.

The prototype is implemented entirely in Java and is available online [20]. While Java provides the benefits of type safety and ease of development, it results in significant performance degradation compared to native code. We believe that our design admits more efficient implementations and plan to demonstrate this in future work.

5.2 Performance

This section present performance results for two applications: *empty*, an empty application to test baseline invocation, and *tax*, a tax preparation application. The experiments use three machines: one TEP server and two participants. Each machine is a 600MHz Intel Pentium III with 512MB RAM. Each runs RedHat Linux 6.0 and Sun's Java Runtime Environment 1.2.2 with a just-in-time compiler. The machines communicate over 100Mb local Ethernet and use 1024-bit RSA keys for public-key cryptography.

5.2.1 Empty

In the *empty* application, an invoker connects to TEP, establishes a secure session, and requests a program by URL. TEP fetches the program, checks it using the standard Java bytecode verifier, and executes the requested method (which is empty).

The total time spent at the invoker is 185ms: 148ms on cryptography for the secure session and 37ms communicating with TEP. TEP spends 148ms on crypto and 35ms handling the request: 25ms waiting for the invoker to complete the session protocol and send its request; 10ms fetching, loading, and verifying the program; and 0ms executing the requested method.

The vast majority of this time is spent on cryptography. Each secure channel creation requires one RSA decryption, encryption, signature, and verification. On the test platform, 1028-bit RSA private key operations take 71ms each and public key operations take 2ms each. The same operations in C++ on a 450MHz Celeron take 27ms and 0.7ms each, an improvement of over 60% [5]. Thus, a native code implementation of cryptography can greatly improve the performance of the system.

5.2.2 Tax

The *tax* application involves two parties: the client (the invoker) and the tax preparer. The application reads income data from the client, opens a channel to the tax preparer, reads data from the preparer, calculates the result, and writes the result to the client. The preparer runs its own agent to accept connections from TEP, create secure sessions, and provide the data needed by the application.

The total time spent at the client is 665ms: 150ms on crypto, 25ms waiting for TEP to finish the session protocol, 60ms waiting for TEP to load and start the application, and 430ms interacting with it. TEP spends 150ms on crypto and 450ms handling the request: 20ms loading the application and 430 ms running it.

The tax application spends 180ms establishing a secure session with the tax preparer and 170ms dynamically generating the policy that binds the preparer's key to its channel name. The remaining 80ms are spent reading data from the participants and writing the result to the client.

A large part of the application time is spent generating a policy using certificate-chain discovery. This time can be eliminated by caching policies ahead of time. Application providers and invokers can provide TEP with the certificate chains needed to prove the required bindings, and thus improve performance.

6 Related Work

6.1 Secure Program Partitioning

The most closely related work to ours is *secure program partitioning*, a technique that protects the privacy and integrity of data in shared computations by splitting programs among multiple hosts [26]. Whereas TEP assumes that there is a single host trusted by all participants, secure program partitioning supports heterogeneous trust of the hosts used in the computation. Both systems take advantage of static information flow analysis and cryptographic techniques to provide privacy and integrity assurances. Both systems currently require that computations be configured statically at invocation, although future work promises greater flexibility.

6.2 Java Applets

Java applets authorize programs by checking that the program is signed by a trusted provider [10]. This form of program authorization requires that the user decide what permissions to grant to code signed by a particular signer and only applies to code that runs on the local machine. In the TEP model, this sort of authorization is required to restrict participants' application-specific agents to accessing the network only through a trusted TEP-provided agent.

TEP's main advantage over the Java model is that a computation does not run locally to any participant in a computation, so no participant gets access to the other participants' data. It does not suffice to run a Java program on an arbitrary third-party server, since the server must provide the appropriate identification and isolation of the computation to ensure that data is not leaked.

6.3 Active Networks

Active networks permit applications to inject programs into networks to deploy new network services [25, 24]. This system shares certain features with TEP: the ability to load new code dynamically and the need to isolate and control that code.

When code is executed at an active node, it must use a restricted API to access the underlying system, much like TEP's channel API. Active nodes use this API to regulate the code's access to shared resources. TEP can be extended similarly to provide certain types of resource control.

6.4 Secure Multiparty Computation

Secure multiparty computation uses cryptographic techniques to run shared computations with minimal or no need for a trusted third party [6, 13]. Such techniques can reduce the amount of trust required of TEP, providing greater assurance for computation participants. However, these techniques often require a majority of honest participants, while TEP does not.

TEP's main advantage over these techniques is flexibility. New applications for TEP are easy to develop, whereas no simple technique exists for developing new secure multiparty applications.

7 Conclusion

We have presented TEP, a trusted third-party computation service. We began by identifying a set of requirements for such services and presented a design that satisfies those requirements while maintaining generality. The design ensures that each participant in a computation has enough information to determine if that computation can leak its data.

We then described an implementation of the design and demonstrated its flexibility with examples. TEP supports programs written in Java that use *channels*, a general mechanism for communicating with the participants in a computation. TEP assigns *roles* to participants to aid in reasoning about computations and supports client-side *agents* that can provide general interfaces to computations on TEP.

We then improved the safety of the system by incorporating static information-flow analysis for TEP programs. TEP accepts programs written in Jif, a Java-based language that can be checked at compile-time for information leaks. We showed how the concepts in the Java version of TEP are represented in Jif and provided a Jif example to demonstrate the changes. Using Jif, TEP can guarantee the privacy of participants in certain computations without requiring that the participants inspect the code.

Finally, we presented the performance for a prototype implementation of TEP. We showed that the prototype can run a simple third-party tax preparation application with reasonable performance. We found that the majority of the time for a computation is spent on the cryptography for creating secure channels. This suggests that optimizing the cryptography and channel protocol can greatly improve performance.

Throughout this paper, we have identified ways in which TEP can be extended. The most fundamental of these is enabling participants to initiate connections to computations on TEP. This allows TEP computations to act as servers for requests from participants. Another important extension is resource control: TEP can protect participants' privacy but currently cannot withstand denial-of-service attacks from the programs it runs. A final extension is support for Jif's runtime mechanisms. These enable programmers to create more general programs for TEP that are guaranteed to protect participants' privacy.

References

[1] S. Ajmani. A Trusted Execution Platform for multiparty computation. Master's thesis, Massachusetts Institute of Technology, July 2000. http://www.pmg.lcs.mit.edu/˜ajmani/papers/thesis.ps.

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proc. of the 6th ACM Conf. on Computer and Communications Security*, Nov. 1999.

[3] D. Balfanz and L. Gong. Experience with secure multi-processing in java. Technical Report 560-97, Princeton University, Sept. 1997.

[4] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. Draft, Nov. 1999.

[5] W. Dai. Speed comparison of popular crypto algorithms. Web Page, Apr. 1999. http://www.eskimo.com/˜weidai/benchmarks.html.

[6] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game, or A completeness theorem for protocols with honest majority. In *Proc. 19th ACM Symp. on Theory of Computing (STOC)*, pages 218–229, 1987.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Aug. 1996. ISBN 0-201-63451-1.

[8] Java 2 platform, standard edition, v1.2.2 API specification. Web Page, 1999. http://java.sun.com/products/jdk/1.2/docs/api/.

[9] The JAR guide. Web Page, 1997. http://java.sun.com/products/jdk/1.2/docs/guide/jar/jarGuide.html.

[10] JDK 1.2 security documentation. Web Page, Apr. 1998. http://java.sun.com/products/jdk/1.2/docs/guide/security/.

[11] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. OOPSLA '98*, volume 33, pages 36–44, Vancouver BC, Canada, Oct. 1998.

ACM Special Interest Group on Programming Languages, ACM Press.

[12] N. Mathewson. Information flow in Java bytecodes. Master's thesis, Massachusetts Institute of Technology, 2000.

[13] S. Micali. Distributed split-key cryptosystem and applications. United States Patent 6,026,163, USPTO, Dec. 1996.

[14] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Jan. 1999.

[15] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, Apr. 2001. http://www.cs.cornell.edu/andru-/pubs.html.

[16] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, Washington, Oct. 1996.

[17] Public-key infrastructure (X.509) (PKIX). Web Page, Feb. 2000. http://www.ietf.org-/html.charters/pkix-charter.html.

[18] M. I. Seltzer et al. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementations (OSDI)*, pages 213–227, Seattle, WA, Oct. 1996.

[19] Simple public key infrastructure (SPKI). Web Page, Feb. 1998. http://www.ietf.org-/html.charters/spki-charter.html.

[20] http://www.pmg.lcs.mit.edu/˜ajmani-/tep/.

[21] Transport layer security (TLS). Web Page, Mar. 2001. http://www.ietf.org-/html.charters/tls-charter.html.

[22] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 203–216. ACM Press, Dec. 1993.

[23] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, pages 116–128, Saint-Malo, France, Oct. 1997.

[24] D. Wetherall, U. Legedza, and J. Guttag. Introducing new Internet services: Why and how. *IEEE Network Magazine*, July 1998. http://www.sds.lcs.mit.edu-/activeware/.

[25] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. IEEE OPENARCH*, San Francisco, CA, Apr. 1998. http://www.sds.lcs.mit.edu/activeware/.

[26] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, volume 35, Banff, Alberta, Canada, Oct. 2001. ACM Operating Systems Review, ACM Press.