Journal of Computer Security 0 (0) 1
IOS Press

Fabric: Building Open Distributed Systems Securely by Construction

Jed Liu *, Owen Arden, Michael D. George, and Andrew C. Myers
Department of Computer Science, Gates Hall, Cornell University, Ithaca, NY, USA
E-mail: {liujed,owen,mdgeorge,andru}@cs.cornell.edu

Abstract. Distributed information systems are prevalent in modern computing but difficult to build securely. Because systems commonly span domains of trust, host nodes share data and code of varying degrees of trustworthiness. Modern systems are often open and extensible, making security even harder to reason about. Unfortunately, standard methods for software construction do not help programmers enough with ensuring their software is secure.

Fabric is a system and language for building open, distributed, extensible information systems that are secure by construction. Fabric is a decentralized system that allows nodes to securely share both data and code despite mutual distrust. All resources are labeled with confidentiality and integrity policies that are enforced through a combination of compile-time and run-time mechanisms.

The Fabric language offers a high-level but powerful model of computation. All resources appear as objects in the language, and the distribution and persistence of code and data are largely transparent to programmers. Fabric supports both data-shipping and query/RPC styles of computation: computation and information can both move between nodes. Optimistic, nested transactions ensure consistency across all objects and nodes. Fabric programs can securely share mobile code across trust domains, enabling more reuse and evolution of code and supporting new kinds of secure applications not possible in other distributed systems. Results from applications built using Fabric suggest that Fabric enforces strong security while offering a clean, concise, powerful programming model with good performance. An open-source prototype is available for download.

1. Introduction

Standard programming abstractions are ill-suited to the task of building modern applications securely, providing little or no assurance that the resulting systems will use information in a secure way. Yet distributed information systems developed using these abstractions are deeply integrated into many activities where security and privacy are crucial. It should be unsurprising that there are so many serious system compromises and privacy violations.

Because current programming languages and abstractions are deeply implicated in the insecurity of modern systems, it suggests that a new programming model is needed for securely implementing distributed systems. But this goal comes with some major challenges. First, before a system can be verified to enforce security requirements, these security requirements must be specified. With current standard

*Corresponding author. E-mail: liujed@cs.cornell.edu

programming models, such specifications are absent, too weak, or too onerous for developers to use. Second, security verification methods are needed. And to be effective, these verification methods should be integrated into the programming model, so that developers receive feedback and guidance as they construct the system. It is not enough to leave security as an afterthought for enforcement at run time—a way is needed to *design* systems to be secure.

This paper describes the Fabric programming system, a clean-slate programming model for building distributed systems that are secure by construction. Fabric is both a programming language and a system that implements this language in a decentralized way. Fabric offers a high-level, object-oriented programming abstraction that differs from typical object-oriented languages such as Java in that information security policies—and some features of the distributed computing environment—are explicitly visible in Fabric code. At compile time, developers can verify that all information flows in their code are secure; at run time, host nodes participating in distributed Fabric computations further check code before running it, to verify that it will enforce information security. An open-source prototype implementation with example programs is available for download [**?**].

This paper offers the first integrated presentation of Fabric. It expands on previous peer-reviewed publications [58,4], and includes details (drawn from dissertations [**? ?**]) on node authentication (Section 6.2), dynamic authorization (Section 6.4), the security cache (Section 6.7), writer maps (Section 6.8), and distributed transactions (Section 6.9). The evaluation results in Section 9 were previously published [**?**].

1.1. Federation and decentralized security

Fabric provides a shared computational and storage substrate implemented by an essentially unbounded number of Internet hosts. As with the Web, there is no notion of an "instance" of Fabric, and there is no centralized control over admission to Fabric. New nodes—even untrustworthy ones—can join the system freely, and begin interacting and sharing information with other nodes without prior arrangement. Interactions between nodes in this decentralized system are made secure by limiting the exchange of information and computation according to the trust that nodes have expressed (or have not expressed) in each other.

Because Fabric is fully decentralized in design, it is able to address an important security challenge: the need for systems that integrate information and computation from independent administrative domains. Such systems are *federated*: each domain has its own security and privacy requirements, but does not fully trust other domains. Integrating information and code in this setting of *heterogeneous trust* is attractive—it enables new services and capabilities including mashups, multi-cloud applications, and distributed web services. Because Fabric does not assume any central authority, it can be used to construct various kinds of federated systems while enforcing security and privacy on behalf of all participants.

To illustrate the challenges of building federated systems securely, consider the scenario of medical institutions that want to securely and quickly share patient information—with each other, with their patients, and with the patients' insurance companies. In fact, incomplete patient information is a leading cause of the more than 44,000 deaths that result annually from preventable medical errors in the United States alone [50]. However, information sharing must respect the security and privacy policies of the two institutions, as mandated by law in many countries. Current information systems lack a principled way to share medical information under verifiable security and privacy guarantees.

We take a holistic view of security: all participants' security requirements must be enforced. But in a federated system, some host nodes are at least partly distrusted by some participants. What does it mean, then, to correctly enforce security? Our guiding principle for security is that one's security should

never depend on components of the system that one does not trust. We call this the *decentralized security principle*. More precisely, it says that the enforcement of a given security policy should not depend on the behavior of any component that is not trusted by the owner of the policy.

Fabric enforces information security through a combination of mechanisms at the language and system levels. To decide whether these mechanisms enforce security in accordance with the decentralized security principle, we need to know what security policies are supposed to be enforced and which host nodes are trusted to enforce which policies. Therefore, Fabric makes both policies and trust explicit in the programming model. With policies and trust made explicit, the decentralized security principle becomes a valuable guide for understanding where enforcement mechanisms are needed, and where they are not.

1.2. Mobile code

Data is exchanged between trust domains rather freely in modern computing systems. So is code. This *mobile code* adds new challenges for making distributed systems secure. Web applications have made mobile code a part of everyday life: visiting a web page typically loads JavaScript code from multiple providers into your browser. In fact, many web pages are already federated systems! Web services such as Facebook allow third parties to provide applications that are dynamically combined with their core functionality. And even traditional desktop applications dynamically download plugins from external providers.

Over the past few years an ecosystem of mobile-code development has sprung up in which web programmers reuse and customize JavaScript code found on the Web for their own purposes. Sometimes programmers simply copy code and modify it for their purposes; web applications commonly import large JavaScript libraries such as JQuery [1] via URLs. Other mobile-code web platforms such as ActionScript, used by Flash, are not fundamentally different. While JavaScript is the most common mobile-code system, studying mobile-code security in the clean-slate setting of Fabric has the advantage of addressing a more general, abstract instantiation of the real problems. The solutions should be correspondingly more reusable.

The benefits of freely sharing mobile code come at a cost: dynamically combining code from multiple sources might yield an insecure application. On the Web now, the main security safeguard is the *same-origin policy* [97], which attempts to limit web applications to communication with their originating website. This policy prevents many useful applications yet also fails to address all the ways that untrusted code can create security vulnerabilities [**?**]. Limitations on expressive power force developers to work around the same-origin policy, potentially introducing additional vulnerabilities [43].

Fabric supports free and flexible distributed sharing of code and data that developers clearly want, while enforcing information security. Confidentiality and integrity of information are protected even when sharing happens between nodes lacking mutual trust. Not only is each individual application component secure, but assemblies of code and data from various providers also satisfy all participants' security requirements.

1.3. The Fabric programming model

Most distributed systems code is currently written using low-level abstractions that obscure application logic, leading to bugs and vulnerabilities. An essential part of the Fabric philosophy is to raise the level of abstraction so programmers can focus on the core application logic, including its security implications.

Central to this abstraction is the representation of many entities as objects that can be used by Fabric code in a uniform way. These objects may be persistent or nonpersistent. They may be stored locally

on the same node, or stored elsewhere in the distributed system. Host nodes, principals, and code are all represented at the language level as objects. This uniform representation allows the same security reasoning and mechanisms to be applied throughout the system.

The Fabric programming language is an extension to the Jif programming language [64,67], which is in turn based on Java 1.4 [36]. Like Jif, the Fabric language has built-in security mechanisms centered around information flow control. In the context of Fabric, these security mechanisms keep untrusted nodes from violating confidentiality and integrity. All objects in Fabric are labeled with policies from an extended version of the *decentralized label model* (DLM) [63]. In this model, security requirements are captured by information flow labels expressed in terms of principals (e.g., users and organizations). Information flow control based on these labels allows principals to control the extent to which other principals (and nodes) can learn about and affect their information. A node that is not trusted by a given principal is prevented from compromising the security policies of that principal.

Despite its high-level programming model, Fabric supports a variety of ways to organize distributed computation and storage for good performance. Fabric supports both *data shipping*, in which data moves to where computation is happening, and *function shipping*, in which computation moves to where data resides. Data shipping enables Fabric nodes to compute using cached copies of remote objects, with good performance when the cache is populated. Function shipping enables computations that span multiple nodes; like database server queries, computations can be shipped to the node where the data is stored persistently. To simplify reasoning about complex computations involving multiple nodes, Fabric provides strong consistency. Inconsistency is prevented by performing all updates within distributed, linearizable transactions.

Of course, there has been much previous work on making distributed systems both easier to build and more secure. Prior mechanisms for remotely executing code, such as CORBA [72], Java RMI [44], SOAP [93] and web services [14], generally offer only limited support for information security, consistency, and data shipping. J2EE persistence (EJB) [29] provides a limited form of transparent access to persistent objects, but does not address distrust or distributed computation. Peer-to-peer content-distribution and wide-area storage systems (e.g., [26,32,53,80]) offer high data availability, but do little to ensure that data is neither leaked to nor damaged by untrusted users. Nor do they ensure consistency of mutable data. Prior distributed systems that enforce confidentiality and integrity in the presence of distrusted nodes (e.g., [103,23,100,21]) have not supported consistent computations over persistent data.

1.4. Threat model

Fabric makes few assumptions about attackers, which yields stronger security assurance. In particular, host nodes in Fabric may be malicious and can behave arbitrarily. They are not assumed to behave in ways allowed by the Fabric programming language, and they need not faithfully follow the distributed protocols by which Fabric nodes communicate. Fabric principals are able to express complete or partial trust in Fabric nodes. If a principal expresses trust in a node, the compromise of that node might harm the security of that principal. The goal of Fabric is to ensure that the degree of trust expressed in a node bounds the degree to which security might be violated from the perspective of that principal.

The runtime exposes more information than what is available at the language level, such as object identifiers and cryptographic keys, which malicious nodes can misuse. However, without the appropriate cryptographic keys, nodes are assumed to be unable to learn encrypted content or forge digital signatures. Because Fabric uses TLS [31] for communication between nodes, we also assume that adversaries cannot tamper with network messages and that they do not learn anything about the contents of network messages unless they control the intended recipient.

A network adversary might use traffic analysis to infer something from the existence of a message, its source and destination, its size, or its timing. We ignore these traffic analysis channels, as do most systems. Fake traffic, predictive mitigation [101], and onion routing [**?**] might mitigate these channels, though at a cost. Network adversaries can also prevent the delivery of messages. The availability of services written using Fabric depends on an assumption that network messages are eventually delivered.

Fabric also does not handle *abort channels*, a side channel arising from the interaction of transactions at different security levels, in which a higher-security transaction can cause a lower-security transaction to abort [**?**]. Abort channels are not observable to programs written in the Fabric language but could be exploited by a malicious node that shares access to an object on which transactions conflict.

Like almost all work on distributed system security, Fabric largely ignores some challenging side channels, arising from timing, termination [83], and progress [6]. Termination and progress channels usually have low bandwidth, but timing channels can have high bandwidth [76]. There are two kinds of timing channels: *external* and *internal* [84]. We do not attempt to control covert external timing channels in this work: adversarial nodes are assumed not to time when messages arrive. Run-time mitigation methods (e.g., [51,101]) may be useful for limiting the bandwidth of external timing channels. Internal timing channels arise when code running within the system measures time, either explicitly or implicitly by constructing a race among concurrent threads. Fabric does not support fine-grained concurrency; a top-level transaction must be sequential. Races between threads therefore involve external communication with a store, and can be considered external timing channels. Resource exhaustion attacks (e.g., by allocating large objects on remote hosts) are also not addressed, although one instance is studied in [**?**].

1.5. Contributions

Fabric integrates many ideas from prior work—including compile-time and run-time information flow, access control, and optimistic transactions—while solving problems that arise from the interactions between these ideas. In fact, it does not seem feasible to provide a high-level programming model like that of Fabric by simply layering existing abstractions. Several new ideas make Fabric possible:

- A programming language that integrates information flow, persistence, transactions, and distributed computation.
- A system that enforces the decentralized security principle by connecting principals and nodes to the security policies (labels) they are trusted to enforce.
- An integration of function-shipping and data-shipping models of computation that enforces secure information flows within and among network nodes.
- A way to manage transactions that affect objects at mutually distrusting nodes while enforcing confidentiality and integrity.
- A method for verifying information flows created by partially trusted mobile code, by integrating the label of the code itself into information flow checking.
- A mobile-code architecture in which partially trusted code becomes a persistent resource that is published and managed by the system.
- Mechanisms for secure evolution of code and code assemblies, and associated persistent data.

Fabric is a mostly clean-slate approach to developing secure systems, and there is work left to be done on interoperability with existing standards. However, Fabric does not require application developers to abandon other standards and methodologies; Fabric can interoperate with other standards. In fact, Fabric already interoperates with existing Java application code, and a library for building web services

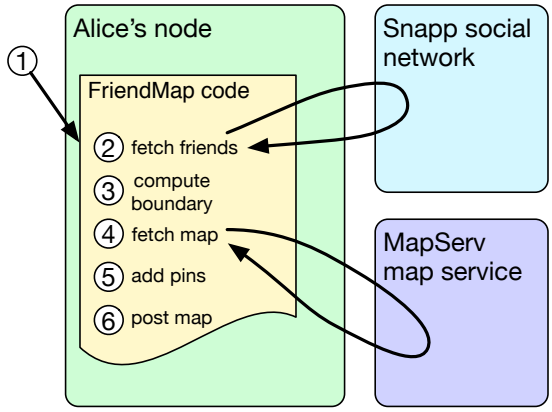


Fig. 1. The FriendMap social mashup

with Fabric already exists. Other standard abstractions and APIs can be implemented using Fabric. It also seems feasible to implement Fabric nodes by encapsulating other services such as databases. Other than its support for language-based security policies, the Fabric programming model should be familiar to users of existing, popular object–relational mapping frameworks such as the Java Persistence API (JPA) [13] or Django [**?**]. It provides transactional access to distributed, persistent objects, with stronger guarantees.

1.6. Overview

The rest of the paper is organized as follows. Section 2 introduces a running example. Section 3 presents the key programming abstractions of the Fabric language. Sections 4 and 5 discuss the language support for information-flow security and secure mobile code. Section 6 describes the design of the Fabric runtime system. Some details of the Fabric implementation are discussed in Section 7. Except for certain features mentioned in Section 7, the design described in this paper has been implemented in an open-source prototype [**?**]. We report on our evaluation of this implementation, including results for the expressiveness of Fabric (Section 8) and the performance of a substantial application built using Fabric (Section 9). Related work is covered in Section 10, lessons learned and future work are discussed in Section 11, and Section 12 concludes.

2. Example

We introduce a running example to illustrate the security challenges faced by applications running in a distributed system with heterogeneous trust. It is an application that we might expect to be able to easily construct in today's web, but cannot. This application, which we call FriendMap, is a mashup that allows a user to create a map displaying the location of their friends. The locations of the friends are obtained from a social network, and the map on which to display these locations is obtained from a mapping service. If Alice or her friends do not fully trust either the social network or the map service, they will have reason to be concerned that the application might violate their privacy.

Figure 1 shows the sequence of events as the FriendMap application is run. First, the FriendMap application code is downloaded and executed (1). The application FriendMap fetches the locations of

Alice's friends (2) from the social network application, which we call Snapp. These locations are used to compute the geographical area that will be displayed (3). Then a bounding box for this area is sent to a mapping service (MapServ), which constructs a map of the area (4). The blank map is returned to Alice's client, and the application places her friends' locations onto the map (5). Alice may then share the annotated map with her friends on the social network (6).

2.1. Security considerations

Even this simple example has complex security requirements because the principals may trust each other to differing degrees. For example, suppose Alice trusts MapServ to learn some information about her friends, but her friend Bob does not trust MapServ at all. In that case, FriendMap must avoid using Bob's location to compute the map request.

Similarly, although Bob trusts Alice to see his location, he may not trust Alice's friends with the same information. If so, FriendMap must either avoid posting the resulting map where Alice's friends can see it or omit Bob's location from the map.

In real applications, policies are more nuanced than lists of entities allowed to learn information. In the FriendMap example, Bob may consider his exact location confidential, but be willing to release some information about where he is, such as the city he is in. Alternatively, he may not mind letting the public know where he was yesterday, but may wish to keep his current location secret.

These more complex policies for allowed information flows can be modeled as a controlled *downgrading* of the information security policy: Bob is willing to *declassify* his location information if its precision is truncated, or if it is more than 24 hours old. In general, downgrading is implemented by application code, and it is critical that the code be authorized to do the downgrading. Any code that declassifies Bob's location must be either provided by or endorsed by Bob.

Enforcing security becomes even more challenging if none of the involved principals trust the provider of the FriendMap code—if the code is downloaded from an untrusted site, much like JavaScript code is in many current web pages. In this case, a mechanism is needed to ensure that the downloaded code enforces all principals' policies; any principal who controls this mechanism or the node on which it operates must be trusted to enforce these policies. In this example, Bob trusts Alice to enforce the confidentiality of his location. Alice's node is therefore responsible for this enforcement, even if the FriendMap code tries to violate it.

2.2. Compositionality of information flow

Standard security mechanisms are not compositional, and therefore fail to meet the security requirements of modern applications, which, like FriendMap, often need to share information across security domains. Isolation mechanisms such as the same-origin policy (SOP) [97] and software fault isolation (SFI) [94] are secure but prevent sharing. For instance, the SOP would prohibit FriendMap from accessing the locations of Alice's friends, since the locations are hosted by Snapp.

Since isolation makes intended functionality impossible, programmers must open channels for communication across isolation boundaries, which can reintroduce vulnerabilities. The next step is to try to control these channels using access control mechanisms such as capabilities. But these attempts seem doomed to fail, because access control is not compositional. Applied to FriendMap, once Alice's client is able to see her friends' locations, access control would not prevent the application from leaking the data to MapServ, her personal bulletin board, or even FriendMap's developers. Access control mechanisms

do not take into account *what* information is being communicated—*any* communication is allowed, as long as it is performed by an authorized principal.

In contrast, information flow control is inherently compositional, making for an appealing security mechanism. When system components that independently enforce information flow policies are combined, the combination also enforces those policies across the whole, as long as policies agree at the interfaces where information crosses between components. Additionally, information flow mechanisms can be proved to ensure strong, extensional security properties such as noninterference and robust declassification [83].

The benefits of compositionality have led to the use of information flow control mechanisms and policies for enforcing confidentiality and integrity in some previous decentralized systems [65,66,98, 103,99,100,82]. However, these systems do not support secure sharing of mobile code, and they provide a lower-level programming interface.

2.3. Software construction and evolution

Modern applications are built from components developed by different organizations and upgraded on different schedules. In a decentralized environment, different principals may trust different software developers and providers, making the challenges even greater. Every user must have confidence that the integrity of the data that they care about is maintained, even if the software manipulating that data comes from an incompletely trusted source.

These requirements conflict with the ability for applications and open services to be upgraded over time. For instance, in the FriendMap example, a service provider like Snapp should be able to update its service to add and remove features and to fix bugs, without coordinating with the open-ended set of application developers using the platform. Similarly, FriendMap should be able to upgrade their software to make use of new features as they become available, without having to coordinate their upgrades with all of the APIs that they rely on.

Components that make up today's software are typically gathered together at least twice: once by the developer for compilation and testing, and once on behalf of the end user for execution. Programs in more dynamic systems—like JavaScript libraries on the Web—may be refetched and reassembled every time a page is displayed. In fact, because such code may be cached, one cannot even be sure that all of the components of the application were fetched at the same time. While this approach allows software to be upgraded piecemeal over time, it can also break assumptions developers have made about the software, potentially creating vulnerabilities.

Fabric provides software interoperability and evolution while enforcing end-to-end information security policies on users' data against any parties these users do not trust. Fabric does this by tracking the provenance of code in the system through integrity labels on the code. Code may also be assigned a confidentiality label by its developers. Such policies allow developers to protect the confidentiality of code and the results computed from it. For instance, the computed results might reveal information about the confidential code itself. Prior information flow systems ignore these kinds of information flows.

3. The Fabric language

The interface between the Fabric system and application developers is the Fabric programming language, a high-level language for building secure distributed programs with mobile code. The Fabric language extends Jif [64,67] with four major features:

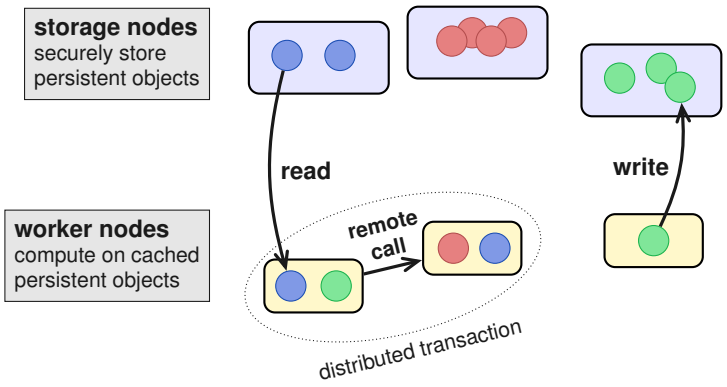


Fig. 2. Fabric architecture

- Nested transactions ensure that computations observe and update objects consistently, and provide clean recovery from failures.
- Remote method calls (remote procedure calls to methods) allow computations that are distributed.
- Remote objects are accessed transparently, as if they are local objects.
- Mobile code allows program components to be dynamically downloaded and executed.

While these features are fairly unusual, individually they are not unique to Fabric. Argus [56] has the first two, for example. The contribution of Fabric is in integrating these features with information flow security. This integration is difficult because new mechanisms are needed to ensure that the underlying implementation of the high-level programming model does not create security vulnerabilities. For example, transactions should not leak confidential information, and remote calls must be properly authorized.

This section presents the key programming abstractions of Fabric, but discussion of language support for information-flow security and for secure mobile code is deferred until Sections 4 and 5, respectively.

3.1. Distributed programming model

Fabric presents to the programmer a two-tier system model, illustrated in Figure 2. The system has two kinds of nodes. *Storage nodes* (or *stores*) are the data repositories in the system. They persistently store data objects and are responsible for the security of those objects. Computation in Fabric is performed by *worker nodes* (or *workers*), using both their own objects and possibly cached copies of objects from other nodes.

Object model Fabric uses objects to represent all code and data in the system, including security principals, information flow labels, remote nodes, local and remote data, and mobile code. Like Java objects, Fabric objects maintain state in mutable fields and have a name that other objects can refer to. But unlike Java, object names in Fabric are not capabilities, because objects have explicit security policies governing their use. Object names can therefore be published outside Fabric—for example, as URLs.

Fabric objects are persistent and distributed: each object is stored on a named host, the storage node of the object. Fabric does not provide *persistence by reachability* [8] because it can lead to unintended persistence. Instead, invocations of class constructors are annotated to indicate the store on which the newly created object should be made persistent. The syntax `new C@s(...)` creates a new object of class `C` on the store identified by the variable `s`. The `@s` annotation may be omitted from constructor calls in instance methods, in which case the new object is created at the same store as the current (`this`) object.

Objects may have non-final fields that are marked `transient`. Transient fields are not saved persistently, which is similar to their treatment by Java serialization [38].

Computational model Unlike most previous languages for distributed programming, Fabric aims to support multiple styles of distributed computation. A single Fabric computation can span multiple workers, access objects from multiple stores, and dynamically download and execute code. This model is more general than web applications, for example, in which computations cannot span multiple clients.

Subject to security constraints, any Fabric worker can use any object, regardless of where the object is stored. When a computation requires an object, a copy of the object is sent to the worker; when the computation completes, updates to the copy are pushed back to the store that holds the authoritative version of the object. Thus, the default mode of computation in Fabric is data shipping, in which data moves to the hosts where computation is happening.

However, Fabric programs can also explicitly transfer execution to a different worker by making a *remote call*. The syntax `o.m@w(...)` indicates that the method `m` of object `o` should be invoked on the worker `w`. Like any other method, `m` can contain arbitrary Fabric code, including remote calls back to the worker calling `m`. We refer to the ability to transfer control to remote workers as *function shipping*. Remote calls in Fabric differ from those in previous language-based RPC mechanisms (e.g., Network Objects [11], CORBA [72], and Java RMI [74]):

- Neither the local calling host nor the remote callee worker `w` needs to have a copy of the object `o` on which the method is invoked. Callee node `w` receives a copy of `o` if it does not yet have one.
- The remote call implicitly starts its own nested transaction, in which the entire method call is executed. The effects of this transaction are not visible to other code running on the remote node until the commit of the top-level transaction containing the nested transaction.
- Remote method calls are subject to authorization checks. The caller checks that the callee is trusted enough to enforce security for the method; the callee checks that the calling node is trusted enough to invoke the method and to see the results of the method. Further details on these checks are deferred to Section 4.7.

One important use of remote calls is to invoke an operation on a worker colocated with a store. Since a colocated worker has low-cost access to persistent data, this kind of remote call can improve performance substantially, and is analogous to database queries and stored-procedure calls in conventional database applications. However, remote calls in Fabric have the advantage that they can run arbitrary Fabric code, which can make further remote calls. Moreover, Fabric enforces information flow security throughout the distributed computation.

Together, function and data shipping can model a variety of architectures used for distributed software. In the FriendMap example, Alice's worker initiates and coordinates the computation. Data shipping allows the worker to use objects stored on Snapp and MapServ. To improve performance, the application makes a remote call to the MapServ worker to generate maps. The data structure representing the resulting map is fetched by Alice's worker as FriendMap processes it for display.

3.2. Transactions

A key challenge for distributed systems is how to provide efficient access to remotely located, mutable data. For low-latency access to data, it is desirable to cache or replicate data widely; however, making sure that the various caches or replicas stay in sync involves relatively expensive distributed protocols. For this reason, many recent distributed storage systems provide only weak consistency guarantees (e.g., [28**?**]).

Despite the recent popularity of weak (e.g., eventual) consistency, we believe that strong consistency is important for building high-level abstractions about which programmers can reason correctly. Weak consistency makes weaker guarantees about what happens when data is read or written, which means programmers have to reason more carefully about whether their code works correctly in all possible program executions. Perhaps as a consequence, exploration of stronger consistency guarantees has been a popular theme of recent research on distributed storage systems (e.g, [**? ? ? ? ? ? ?**]).

Therefore, Fabric provides transactions with strong consistency guarantees; our expectation is that when Fabric programmers actually want weak consistency, such as when building a low-latency geo-replicated system, they will implement weakly consistent replication on top of the strong consistency provided by Fabric. This strategy is likely simpler than trying to build strong consistency above weak consistency, and also offers a cleaner programming model. It may make sense to integrate some form of weaker consistency into Fabric, but we leave this to future work.

Transactions are indicated in the Fabric language by marking blocks of code `atomic`, or by making remote calls. Each transaction is executed atomically and in isolation from all other computations in Fabric. It appears to commit instantaneously, and all later transactions should observe the same or newer state. In other words, committed Fabric transactions are strictly serializable [77] or, equivalently, linearizable [41].

Linearizability allows programmers to program as if theirs is the only program in the world that is executing for the duration of a transaction. Reasoning about correctness is greatly simplified when programmers can rule out interference from concurrent computations accessing the same data. Linearizable transactions are also important for security: our information flow analysis relies on the fact that when a program dynamically performs label comparisons, the results of that comparison are meaningful. Linearizability ensures that the results of security-critical label comparisons and dynamic authorization checks are correct at the time they are performed by the transaction, and remain correct during the remainder of the transaction as long as the transaction does not itself change the outcome of these checks by modifying trust relationships among principals.

To cleanly handle application-defined failures, transactions may be aborted by exceptions. If a transaction body throws an exception, the transaction is considered to have failed, and is aborted by rolling back its side effects. Failure due to conflict with other transactions causes the atomic block to be retried automatically with exponential back-off.

Transactions can be nested, so atomic blocks may be used even during a transaction. Not only do nested transactions allow for finer-grained rollback, but they also make code compositional: programmers can enforce atomicity at any layer of abstraction within a system. In contrast, transactions that do not support nesting can only be used at the "top level" of the program. Object fields may be read outside of transactions, but updates must occur within transactions; a field update that occurs outside a transaction is automatically wrapped in its own top-level transaction.

Computing with remote calls results in a transaction that spans all workers involved, with each remote call in its own subtransaction. The Fabric runtime system ensures that when multiple workers use the same object within a transaction, updates to the object are propagated between them as necessary.

Individual transactions are single-threaded; new threads cannot be started inside a transaction. This choice was made largely to simplify the implementation, though it maps well onto the OLTP-style[1] applications that Fabric targets. We leave multithreaded transactions to future work.

[1]OLTP (Online Transaction Processing) applications handle online transaction requests from external clients.

However, Fabric workers are multithreaded and can concurrently serve remote-call requests from other workers. Applications can also start transactions that run concurrently. To increase the effective size of the worker's cache, these concurrent transactions share access to the same underlying cached objects. Interference between concurrent transactions at the same worker is prevented using reader–writer locks.

3.3. Implementing the FriendMap example

The features of the Fabric language can significantly simplify code. As an example, consider the FriendMap application described in Section 2. Atomic transactions and transparent distribution allow its code to be written in a way that is very similar to what would be written for a nondistributed, nonpersistent version of the same application.

The core computation of the application needs to aggregate all the locations of the user's friends into a bounding box. Ignoring security concerns for the moment, this computation can be written very simply, even though it accesses data (the friends' locations) that is stored at a remote node. No explicit communication or data conversion is needed:

```
Box boundary = new Box(0,0,0,0);
for (User friend : user.friends)
    boundary.expand(friend.location);
```

Once the boundary has been computed, a remote call is used to ship this data to the map service, which runs on a worker node referenced by the variable `ms`:

```
Worker ms = ... // set to the MapServ worker node
Map map = mapService.getMap@ms(boundary);
```

Finally, we augment the map with the friends' locations. This computation also transparently uses remotely stored information:

```
for (User friend : user.friends)
    map.addPin(friend.location);
```

This code is much shorter and simpler than the equivalent code that would be needed for other distributed computing frameworks.

However, the code as written is not yet secure: additional checks must be added to the code to make sure that the friends accept having their location leaked to the map service, and that they are okay with having their location marked with a pin in the application output. In Section 4.9, we show how the security aspects of the Fabric language force the programmer to add dynamic checks that ensure the friends' privacy is in fact not violated.

3.4. Java interoperability and FabIL

Fabric applications can be written using a mixture of Java, Fabric, and FabIL (the Fabric intermediate language). FabIL is an extension to Java that supports transactions, remote calls, and access control. A key difference between Fabric and FabIL is that FabIL does not enforce information flow security.

More concretely, FabIL supports the features described in this section: it provides `atomic` blocks, supports the syntax `new C@s(...)` for constructing persistent objects on stores, and gives the ability to

make remote calls with the syntax `o.m@w(...)`. Transaction management is performed on Fabric and FabIL objects but not on Java objects, so the effects of failed transactions on Java objects are not rolled back. Additionally, objects created in FabIL are equipped with a programmer-specified access control policy for protecting the object at run time.

FabIL and Java code is considered trusted, and workers only execute trusted code that is stored on their local file system. This design is compatible with the decentralized security principle because the effects of trusted FabIL and Java code are confined to principals that already trust the nodes running the code.

FabIL can be convenient for code whose security properties are not accurately captured by static information-flow analysis, making the labels of the full Fabric language inconvenient. One example is code implementing cryptography, where the annotation burden of labels is not worth the cost; a second example is the code implementing internals of Fabric, such as its built-in label and principal objects.

4. Information flow security

Fabric allows information security policies to be expressed directly within programs. It uses information flow control to protect the confidentiality and integrity of information flowing through Fabric programs and across the Fabric system. *Principals* and *labels* describe the security requirements on data within the system, enabling reasoning about whether programs are secure. Many aspects of the model and language are inherited from Jif, but Fabric adds a new *trust ordering* on labels. Fabric's language also has novel features to enforce the security of remote calls and to prevent side channels caused by its distributed implementation.

4.1. Background

Principal model Like Jif, Fabric uses abstract *principals* to represent entities that can trust or be trusted, such as users, roles, groups, organizations, and privileges (and Fabric nodes). In the FriendMap example, `alice`, `bob`, `snapp`, `friendmap`, and `mapserv` are all principals.

Principals express their trust relationships through the *acts-for* relation [66]. If a principal p acts for a principal q (written $p \succcurlyeq q$), then q trusts p completely, and q is said to *delegate to* p. The principal p may perform any action that q may perform. For example, p may read any data that q can read, update any data that q may update, or downgrade policies that q can downgrade. The acts-for relation is transitive and reflexive, making it a preorder. We refer to the set of principals under the acts-for relation as the *principal hierarchy*.

There is a *top principal* $\top$ that acts for all principals,[2] and all principals act for the *bottom principal* $\bot$. The operators $\wedge$ and $\vee$ can be used to form conjunctions and disjunctions of principals. The *conjunctive principal* $p \wedge q$ represents the joint authority of p and q, and acts for both: $p \wedge q \succcurlyeq p$ and $p \wedge q \succcurlyeq q$. The *disjunctive principal* $p \vee q$ represents the disjoint authority of p and q, and is acted for by both: $p \succcurlyeq p \vee q$ and $q \succcurlyeq p \vee q$.

While the acts-for relation represents complete trust, programs can create auxiliary principals to represent finer-grained notions of trust. Delegations from these auxiliary principals encode which other principals are sufficiently trusted to carry out certain actions. For example, each user principal in the FriendMap application has an associated `friends` principal that can be used in the specification of poli-

[2]The top principal does not correspond to any physical entity. Indeed, there is never really a top principal, since the maximum authority at a given node is that node itself.

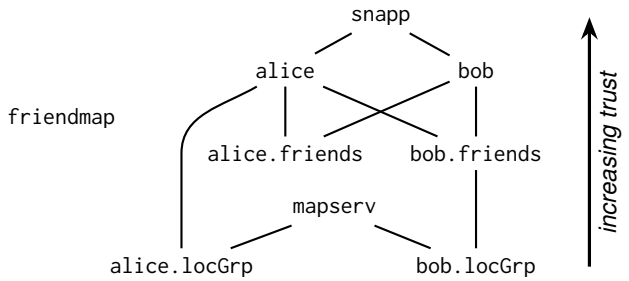


Fig. 3. Trust relationships in the FriendMap example.

Alice's friends list:	$\{\texttt{alice} \leftarrow\}$
Bob's location:	$\{\texttt{bob} \rightarrow \texttt{bob.locGrp}; \texttt{bob} \leftarrow\}$
Requests to MapServ:	$\{\top \rightarrow \texttt{mapserv}\}$
MapServ responses:	$\{\top \leftarrow \texttt{mapserv}\}$

Fig. 4. Labels in the FriendMap example

cies. As shown in Figure 3, the principal `alice.friends`, representing Alice's group of friends, delegates to Alice and each of her friends (e.g., Bob). However, Alice does not delegate to her group of friends; this allows her to retain privileges that she does not extend to them. Additionally, each user has a `locGrp` principal that delegates to those principals that are trusted with the user's location. For example, Alice doesn't trust her friends to keep her location secret, so $\texttt{alice.friends} \not\geqslant \texttt{alice.locGrp}$, whereas Bob trusts his friends with his location ($\texttt{bob.friends} \geqslant \texttt{bob.locGrp}$).

Label model Fabric programs express security policies using *labels*, which are drawn from the decentralized label model (DLM) [66]. Each label is a set of policies that are expressed in terms of principals. Labels are *decentralized:* each policy is owned by a principal, and the policies in a label can have distinct owners. This keeps track of whose security is being enforced, which is useful for Fabric, where principals need to cooperate despite mutual distrust.

For example, Bob might protect information about his location with the *confidentiality policy* $\texttt{bob}\rightarrow\texttt{bob.locGrp}$, which states that Bob controls the policy on the data, but he allows the data to flow to the `bob.locGrp` principal (and implicitly to himself[3]). Similarly, Alice's timeline on the social network might have the *integrity policy* $\texttt{alice}\leftarrow\texttt{alice.friends}$, meaning that Alice owns her timeline, but allows her friends (and implicitly Alice herself) to post to it. Her friend list, however, might have the policy $\texttt{alice}\leftarrow$, indicating that she only allows herself to modify the list.[4]

A label is written as a set of confidentiality and integrity policies, such as $\{\texttt{bob}\rightarrow\texttt{bob.locGrp};\texttt{bob}\leftarrow\}$. A principal can learn about a value only when that principal satisfies *all* confidentiality components in the value's label, and can influence a value only when it satisfies *any* integrity component in the label. Figure 4 gives some of the labels in the FriendMap example. Requests to the map service are labeled $\{\top\rightarrow\texttt{mapserv}\}$, indicating that everyone ($\top$) agrees that MapServ is permitted to read those requests. Similarly, responses from the map service have a label indicating that everyone agrees that MapServ has

[3]The policy $\texttt{bob}\rightarrow\texttt{bob.locGrp}$ is therefore equivalent to $\texttt{bob}\rightarrow\texttt{bob.locGrp} \vee \texttt{bob}$.
[4]The policy $\texttt{alice}\leftarrow$ is short for $\texttt{alice}\leftarrow\texttt{alice}$.

affected those responses. We use $C(\ell)$ and $I(\ell)$ to mean the confidentiality and integrity halves of a label ℓ, respectively.

Fine-grained principals like a user's friend group enable dynamic control over the meanings of policies by changing the principal hierarchy. For example, in Figure 3, Bob has allowed all of his friends to read his location by making the `bob.locGrp` principal delegate to `bob.friends`. When Bob makes a new friend, say with Carol, then Carol will be able to learn Bob's location once he adds the delegation `carol` $\succcurlyeq$ `bob.friends`.

Flows-to relation Labels are part of types in the Fabric language. The label in a program variable's type bounds the information that has affected the value of the variable. As part of type checking at compile time, *label checking* ensures that the information flows in the program are safe according to an *information flow ordering* on labels. When flow from label ℓ_1 to label ℓ_2 is secure, we say that ℓ_1 *flows to* ℓ_2 and write $\ell_1 \sqsubseteq \ell_2$. The flows-to relation is derived from the acts-for relation $\succcurlyeq$.

Confidentiality policies allow public data to affect secret data. For example, we have `alice`$\rightarrow$`bob` $\sqsubseteq$ `carol`$\rightarrow$`dora` exactly when the new owner acts for the old owner (`carol` $\succcurlyeq$ `alice`) and the new readers act for the old readers (`dora` $\succcurlyeq$ `bob` $\vee$ `alice`). Integrity works the opposite way, because integrity policies allow flows from trusted sources to untrusted recipients: the relationship `alice`$\leftarrow$`bob` $\sqsubseteq$ `carol`$\leftarrow$`dora` holds exactly when `alice` $\succcurlyeq$ `carol` and `bob` $\succcurlyeq$ `dora` $\vee$ `carol`. These rules are justified in more detail elsewhere [63].

The following code illustrates these rules. The assignment from `y` to `x` (line 3) type-checks and is secure because the information in `y` can be learned by fewer readers (only Bob rather than both Bob and Carol). The assignment from `x` to `y` (line 4) is rejected by the compiler, because it permits Carol to read the information. However, the second assignment from `x` to `y` (line 6) is allowed because it occurs in a context where Carol is known to act for Bob, and can therefore already read any information that Bob can.

```
1 int {alice→bob} x;
2 int {alice→bob∨carol} y;
3 x = y; // OK: bob ≽ (bob ∨ carol)
4 y = x; // Invalid
5 if (carol ≽ bob) {
6   y = x; // OK: carol ≽ bob, so (bob ∨ carol) ≽ bob
7 }
```

The flows-to relation is a preorder on labels. We can construct an *information flow lattice* by defining an equivalence relation $\ell_1 \equiv_\sqsubseteq \ell_2 \Leftrightarrow (\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1)$, and taking equivalence classes to be lattice elements. We refer to joins and meets in this lattice as *flow joins* and *flow meets*, respectively. The least label $\{\top\leftarrow\}$ describes information that can flow anywhere, because it is public and completely trustworthy. The greatest label is $\{\top\rightarrow\}$, which describes information that can flow nowhere, because it is completely secret and completely untrustworthy.[5]

Implicit flows Information can be conveyed by program control flow. If not controlled, these *implicit flows* can allow adversaries to learn about confidential information from control flow, or to influence high-integrity information by affecting control flow.

[5]The label $\{\top\leftarrow\}$ is equivalent to $\{\bot\rightarrow;\top\leftarrow\}$, because policies owned by $\bot$ can be dropped. Similarly, $\{\}$ and $\{\top\rightarrow\}$ are equivalent to $\{\bot\rightarrow;\bot\leftarrow\}$ and $\{\top\rightarrow;\bot\leftarrow\}$, respectively.

Fabric controls implicit flows through the *program-counter label*, written pc, which captures the confidentiality and integrity of control flow. The program-counter label works by constraining side effects; to assign to a variable x with label L_x, Fabric requires $pc \sqsubseteq L_x$. If this condition does not hold, either information with a stronger confidentiality policy could leak into x, or information with a weaker integrity policy could affect x.

Implicit flows cross method-call boundaries, both local and remote. To track these flows, object methods are annotated with a *begin label* that constrains the program-counter label of the caller, as well as the effects of the method. The pc of the caller must flow to the begin label, which in turn must flow to the label of any variables assigned by the method. This ensures that the caller's pc can flow to the method's assignments. Implicit flows via exceptions and other control-flow mechanisms are also tracked [64].

Authority and downgrades In general, the flows-to relation $\sqsubseteq$ only *approximates* the true information security requirements of an application; sometimes it prevents flows that applications need. Like other systems with information flow control, Fabric allows these flows using *downgrading* operations. Declassification is a downgrading operation that reduces confidentiality; endorsement is one that boosts integrity.

Downgrading can be dangerous to security, so the syntax of Fabric makes all declassification and endorsement explicit. Additionally, all policy-downgrading code must possess the *authority* of a principal that authorizes the downgrade. This principal must act for the owner of any policy that is weakened. Further, declassification and endorsement may only happen in code whose control flow is unaffected by low-integrity information. This restriction enforces *robust downgrading* [5], which prevents the adversary from causing these operations to be misused.

Code can obtain the authority of a principal p in two ways. First, a class can declare it has the authority of p by using a clause `authority(`p`)`. A method of the class can then claim this authority with a clause `where authority(`p`)`. Second, authority can be delegated via a method call, if the method being called is annotated with a clause `where caller(`p`)`. A method with authority delegated in this way can be called only from code that possesses the authority of p. While this model of delegating authority has similarities to Java stack inspection [95], it differs in that authority is statically checked except at remote method calls, where the receiver checks that the calling node is sufficiently trusted to act for the claimed authority.

4.2. Principal objects

Principals have both identity and state (such as their delegation information), so at the language level, Fabric represents principals as objects that may be persisted on stores. If the FriendMap program refers to `alice` or `bob`, these are program variables that refer to specific principal objects, so the names `alice` and `bob` can refer to the same principals no matter where the program is running. All Fabric principals are instances of the Fabric class `fabric.lang.security.Principal` or of one of its subclasses. Applications may define their own principal classes, for example to implement application-specific authentication mechanisms.

Fabric nodes are also principals. At the language level, a Fabric node is a first-class object that can represent the node as a principal in labels and security checks. For example, a storage node might be represented as a variable `s` of type `fabric.worker.Store`. The expression `s actsfor p` tests whether a principal `p` trusts `s`.

Principals control their acts-for relationships by implementing a method `p.delegatesTo(q)`, which tests whether `q` directly acts for `p`. This method allows a principal to say who can directly act for it. In

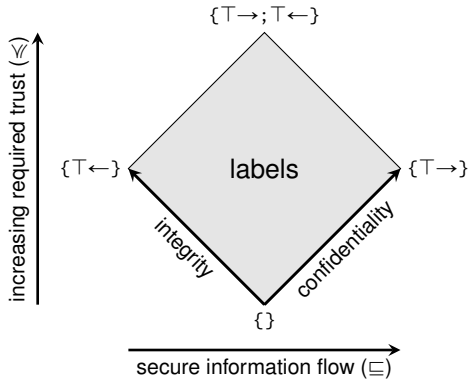


Fig. 5. Orderings on the space of labels

addition, a principal is always considered to delegate to the store at which the principal object is stored. The Fabric runtime system at each worker node automatically computes the transitive closure of the direct acts-for relationships computed by `delegatesTo`.

In general, worker nodes may have different partial views of the acts-for relation. The monotonicity of the label system ensures that security decisions based on these partial views are sound: adding more acts-for relationships to a worker's partial view would only make the worker's security decisions more permissive [66].

4.3. Trust ordering

Fabric extends the DLM by defining a second ordering on labels, the *trust ordering*, which is also derived from the acts-for relation. This ordering is useful for reasoning about the enforcement of policies by a partially trusted platform. A label ℓ_1 may require at least as much *trust* as a label ℓ_2, written $\ell_1 \succcurlyeq \ell_2$ by analogy with the trust ordering on principals. If ℓ_1 requires at least as much trust as ℓ_2, then any platform trusted to enforce ℓ_1 is also trusted to enforce ℓ_2. This happens when ℓ_1 describes confidentiality and integrity policies that are at least as strong as those in ℓ_2; unlike in the information-flow ordering, integrity is not opposite to confidentiality in the trust ordering. Therefore, both confidentiality and integrity use the same rules in the trust ordering: both `alice`→`bob` $\succcurlyeq$ `carol`→`dora` and `alice`←`bob` $\succcurlyeq$ `carol`←`dora` are true exactly when `alice` $\succcurlyeq$ `carol` and `bob` $\succcurlyeq$ `dora` $\vee$ `carol`. The trust ordering is a preorder. We can lift it to a partial order just as we did for flows-to, with equivalences defined by $\ell_1 \equiv_{\succcurlyeq} \ell_2 \Leftrightarrow (\ell_1 \succcurlyeq \ell_2$ and $\ell_2 \succcurlyeq \ell_1)$, which induces a *trust lattice* with a *trust join* and a *trust meet*.

Figure 5 depicts how the two label orderings relate. In the diagram, information flows rightward and required trust increases upward. The least label {⊤←} and the greatest label {⊤→} in the information flow ordering are shown as the leftmost and rightmost elements, respectively. In the trust ordering, the least label (at the bottom of the diagram) describes information that requires no trust to enforce its security, because it is completely public and completely untrustworthy: {}. The greatest label, at the top of the diagram, is for information that is maximally secret and trustworthy: {⊤→;⊤←}. Recent work on Flow-Limited Authorization [**?**] explores the trust ordering further by unifying principals and labels as elements of the same lattice. In Fabric, however, principals and labels remain distinct entities.

4.4. Enforcing labels

Every Fabric object has an associated *update label* that summarizes the trust needed to enforce the object's security. Because the security requirements of every field must be enforced, the update label is the trust join of the labels of the object's fields. The update label acts as an access control policy, determining which nodes can securely read, write, and persistently store the object. An object may be securely stored on a storage node only if the node is trusted to enforce the update label. A worker wishing to read and directly compute on an object only needs to be trusted to enforce the confidentiality half of the update label; to update an object, a worker must be trusted to enforce the integrity half.

Recalling that nodes are principals, we say a principal p *enforces* a label ℓ, written $p \succcurlyeq \ell$,[6] if p can both see and affect data at that label. This is captured formally using the trust ordering on labels:

$$p \succcurlyeq \ell \stackrel{\triangle}{\iff} \{\top \to p; \top \leftarrow p\} \succcurlyeq \ell \tag{1}$$

To see this, suppose ℓ has a confidentiality policy $\{a \to b\}$, which is equivalent to $\{a \to a \vee b\}$. Condition 1 holds exactly when $\top \succcurlyeq a$ (always true) and $p \succcurlyeq a \vee b$. One of the two readers a and b must trust p. Similarly, if ℓ has an integrity policy $\{a \leftarrow b\}$, we require the same condition, $p \succcurlyeq a \vee b$.

In the FriendMap example, if Snapp were untrustworthy, it could release the locations of the users to third parties. This is a risk that the users are willing to take. In Figure 3, Bob designates this by declaring `snapp` $\succcurlyeq$ `bob.locGrp`, allowing Snapp to obtain Bob's location object, according to the labels in Figure 4. On the other hand, the users are not willing to believe that the FriendMap application provider is trustworthy. For example, `friendmap` $\not\succcurlyeq$ `bob.locGrp`, which prevents FriendMap from obtaining Bob's location object.

Taking the trust join of the fields' labels is a conservative choice for the object's update label. As part of our recent work on abort channels [**?**], the compiler in the latest Fabric release [**?**] recovers per-field precision by performing *object splitting*—a level of indirection is added to the object graph, so that the fields of each object in the system have the same label.

4.5. The decentralized security principle

The decentralized security principle says that security of a principal does not depend on any part of the system that it considers to be an adversary. However, who the adversary is depends on whom you ask. From a principal p's perspective, the adversary is some untrusted principal $A \not\succcurlyeq p$. In the FriendMap principal hierarchy in Figure 3, Alice and Bob both treat the FriendMap application provider as an adversary since they don't trust it. Similarly, Snapp and MapServ do not delegate to anyone, so they consider everyone to be adversaries. For simplicity, we assume that Alice and Bob delegate to Snapp, so Snapp is not an adversary for them.

The system should be secure from the perspective of every participating principal, so any principal may be considered an adversary. Fabric's security mechanisms are designed to enforce security regardless of the choice of adversary. Therefore, we analyze security with respect to arbitrary adversary principal A.

Our goal is to allow secure interaction despite distrust, so adversaries are able to read and update certain objects stored at trustworthy nodes. A node n allows a node A to read an object with a low

[6] This is notationally consistent with the acts-for relation and the trust ordering, because transitivity still holds when the three relations are combined: if $p \succcurlyeq q \succcurlyeq \ell' \succcurlyeq \ell$ then $p \succcurlyeq \ell$.

confidentiality label ℓ; that is, if $A \succcurlyeq C(\ell)$. Similarly, A can update objects with low integrity labels; that is, if $A \succcurlyeq I(\ell)$. Objects that represent mobile code are particularly important examples of objects that adversaries might affect. Adversaries can provide mobile code to trustworthy nodes, as long as the code has a low integrity label. Section 5 describes mechanisms that prevent this code from compromising security.

In a decentralized system like Fabric, there is no global *trusted computing base* (TCB). In fact, the decentralized security principle generalizes TCBs, because each label ℓ has its own trusted computing base, consisting of the enforcement mechanisms on nodes n where $n \succcurlyeq \ell$. The decentralized security principle is also more precise than TCBs, since it defines *which* security policies ℓ may be violated if some set of components is compromised.

4.6. Access labels

When following a reference to a Fabric object, the worker may need to fetch the object from its store. This means that the store may learn that the worker is accessing the object, which may reveal information about the pc at the point of dereference. We refer to this kind of covert channel as a *read channel* [98].[7]

The update label does not prevent read channels since it only protects the information in the object itself. To prevent read channels, we need to protect information in the context that is accessing an object. In other words, we need to ensure that the store of the object is permitted to see the pc. This means that the compiler needs some static representation of what store will be contacted when a remote reference is followed.

This representation appears in the language as a second, confidentiality-only label on each object field, called the *access label*. It ensures that the field is stored on a node that is trusted to learn about all the accesses to it, and it prevents the field from being accessed in a context that is too confidential. The access label has no integrity component—Fabric does not consider an integrity dual to read channels, though enforcement of referential integrity in a distributed system may lead to such a dual [**?**].

A field's access label is written as part of the field's type declaration. A label annotation ℓ_u`@`ℓ_a on a field indicates that the field has the update label ℓ_u and the access label ℓ_a, meaning that updates to the field are protected by ℓ_u and that the field's value is hosted by a store that enforces confidentiality ℓ_a. Just as there is an object-level update label that summarizes the update labels of an object's fields, there is also an object-level access label, defined as the join of the access labels of the fields.[8]

Access labels require two static checks in Fabric code. First, to prevent accesses from leaking information, all reads and writes of fields must occur in a context whose label flows to the access label. Specifically, the program-counter label pc must flow to the access label ($pc \sqsubseteq \ell_a$) at each field access (read or update). This is in addition to the implicit-flow check, mentioned in Section 4.1, that requires $pc \sqsubseteq \ell_u$ at each update.

Second, an object may only be allocated on a store trusted to enforce the object's access label. To allocate at a node n an object with access label ℓ_a requires $n \succcurlyeq \ell_a$, because node n will observe future accesses to the object.[9] Together, the two static checks $pc \sqsubseteq \ell_a$ and $n \succcurlyeq \ell_a$ ensure that the node storing the object can safely learn about all accesses to the object: at each access, $pc \sqsubseteq \{\top \rightarrow n\}$.

[7] While *read channel* is the name used in the literature, it refers to information that can be leaked by any observable access to objects, including both reads and writes.

[8] Both the flow and the trust joins coincide here, since access labels do not have integrity components.

[9] The compiler additionally requires that $pc \sqsubseteq \{\top \rightarrow n\}$ at the point of allocation, because node n observes the allocation.

```
1  void m1{alice←} () {
2    RemoteWorker rw = findWorker("bob.example.org");
3    if (rw ⩾ bob) {
4      int{alice→bob} data = 1;
5      int{alice→} y = m2@rw(data);
6    }
7  }
8
9  int{alice→bob} m2{alice←} (int{alice→bob} x) {
10   return x+1;
11 }
```

Fig. 6. A remote call in Fabric

For example, the following class contains public information in a field `data`; accesses to this field are visible to a node `n`:

```
class Public {
    int {} @ { ⊤→n } data;
}
```

Even though the information is public and untrusted (update label `{}`), objects of this class can be stored only on nodes that are at least as trusted as `n`. Then, given a reference `o` to an instance of `Public`, the expression `o.data` will type-check only if the pc flows to `{⊤→n}`. If we had instead given the field `data` the annotation `{}@{}`, the object could be stored on any node, but the type system would prevent accesses from non-public contexts. In other words, the expression `o.data` would type-check only if pc is public.

4.7. Secure remote calls

Fabric programs use remote calls to explicitly transfer execution between workers. Figure 6 shows an example in which a method `m1` dynamically looks up a remote worker `rw` using its hostname and, in line 5, transfers control to `rw` to call a method `m2` on the current (`this`) object. The two workers participating in a remote call have security concerns regarding the information sent and received during the call. These concerns are satisfied with four kinds of checks, illustrated in Figure 7. The checks are made *before* the receiving worker executes the call, and relate the calling worker `cw`, the receiving worker `rw`, the set L_c of labels on the information sent by `cw`, and the set L_r of labels on the information returned by `rw`.

The left side of Figure 7 shows that the compiler permits a remote call only if it can determine that the call is secure from the perspective of the caller. This requires that the receiving worker `rw` pass two kinds of *caller-side checks*. First, `rw` should enforce the confidentiality of the information sent in the call: `rw` $\geqslant C(\ell)$ for every $\ell \in L_c$. The contents of L_c is taken from the signature of the method being called, and includes the begin label and the labels on the method parameters. The worker receiving the call should also enforce the integrity of the information returned, so the second kind of caller-side check is `rw` $\geqslant I(\ell)$ for $\ell \in L_r$. The set L_r is also taken from the method signature, and includes the label on the return value and on any exceptions thrown.

For example, in Figure 6, the method parameter `x` on line 9 is labeled `{alice→bob}`, so the method `m2` can be called on `bob` only because the call happens in a context where it is known that `rw` $\geqslant$

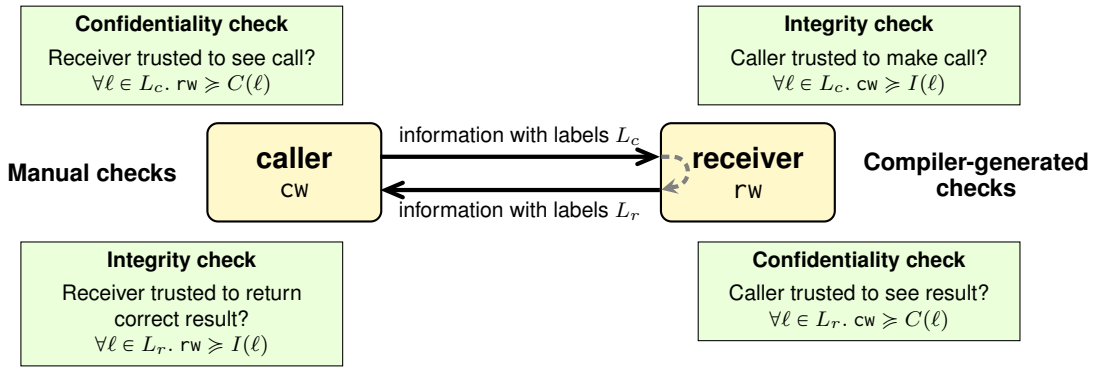


Fig. 7. Manual and compiler-generated checks for remote calls

{alice→bob}. This is established by the test rw $\succcurlyeq$ bob on line 3. Without this test, the compiler would reject the program.

The right side of Figure 7 shows the two kinds of *receiver-side checks* that rw makes when receiving a call. These are analogous to the caller-side checks: the calling worker cw should enforce the integrity of all information that it sends (cw $\succcurlyeq I(\ell)$ for $\ell \in L_c$), and cw should enforce the confidentiality of the information returned (cw $\succcurlyeq C(\ell)$ for $\ell \in L_r$). In Figure 6, the method m2 has a begin label that requires the integrity {alice ←}. Therefore, when bob.example.org receives the remote call to m2, it will check that the calling worker enforces that integrity: cw $\succcurlyeq$ {alice←}. The compiler generates code for these receiver-side checks, and additionally ensures that the checks themselves do not leak information.

4.8. Revocation

Fabric supports the revocation of acts-for relationships. Revocation creates a trade-off between security and performance (or availability), often encountered in the design of public-key infrastructures [90]. To make sound authorization decisions, revocations should rapidly propagate to all who might rely on the revoked authority. However, rapid propagation comes at a performance cost, and the propagation mechanism itself can be vulnerable to denial-of-service attack.

Fabric does not guarantee immediate notification of revoked acts-for relationships, so revocations generally do not take immediate effect. Suppose a principal p keeps the state of its delegations in an object d. A worker that has cached d will not see a revocation of "q acts for p" until it receives an updated copy of d. The worker might not receive this update until a transaction attempts to commit, after having made authorization decisions based on the revoked acts-for relationship. For instance, the worker might allow a remote call to another worker that is no longer trusted, creating flows of information that would not otherwise be permitted with an up-to-date copy of d. This has implications for confidentiality: the remote worker might no longer be permitted to learn the information embodied in the remote call. Integrity is maintained, however, because the transaction will be rolled back and retried with the new version of d, just like any other transaction that observes stale data. Confidentiality guarantees could be improved, at the cost of increased latency, by immediately checking the freshness of the objects used by the transaction before each remote call, but this is not currently done in Fabric.

```
1  Map{resultL} createMap{friendAccess}(User user, label resultL, label friendAccess)
2  where
3    { provider ⊔ MapService.provider ⊔ friendAccess ⊔ {⊤←user} }
4        ⊑ { resultL ⊓ { ⊤→user .store} ⊓ { ⊤→ms } } ,
5    resultL ⊑ { ⊤→user .store}
6  {
7    // Compute a bounding box containing user's friends.
8    label fetchLabel = {resultL ⊓ {⊤→ms}};
9    Box boundary = new Box(0,0,0,0);
10   for (User friend : user.friends)
11     if (friendAccess ⊑ {⊤→friend .store}
12         && {friend→friend.locGrp} ⊑ fetchLabel)
13       boundary.expand(friend.location);
14
15   // Get a local copy of the map from the map service.
16   Map map = mapService.getMap@ms(boundary);
17
18   // Add friends' locations to map.
19   for (User friend : user.friends)
20     if (friendAccess ⊑ {⊤→friend .store}
21         && {friend→friend.locGrp} ⊑ resultL)
22       addPin(map, friend.location, friend);
23   return map;
24 }
```

Fig. 8. An important part of the FriendMap code. Some details have been changed for clarity (e.g., Fabric does not currently support Java's enhanced `for`-loop syntax, and some error handling has been elided).

4.9. Revisiting the FriendMap example

We now revisit the key code from the FriendMap example of Sections 2 and 3.3, to illustrate how Fabric's security features help the programmer write secure code. Recall that the application runs on Alice's worker, and integrates code from FriendMap, MapServ, and Snapp with data from Snapp and MapServ. Figure 8 shows the entirety of the `createMap` method, which provides the key functionality of the application. The code of this method, without any security annotations or checks, was previewed in Section 3.3. As before, the method computes a bounding box of a user's friends (lines 8–13), uses that bounding box to fetch an image from MapServ (line 16), and then annotates that map with the user's friends' locations (lines 19–22).

The method takes the dynamic labels `resultL` and `friendAccess` as arguments. The `resultL` argument describes the policy on the created map, and is used in two flows-to tests to ensure that private locations of friends will not affect the resulting map. One test is on line 12 (where `resultL` is part of `fetchLabel`) and another is on line 21. Each friend's location is only viewable by principals in that friend's location group, specified by the dynamic principal `friend.locGrp`. If the result label is too permissive for a particular friend (i.e., if `{friend→friend.locGrp}` $\not\sqsubseteq$ `resultL`), then the two tests fail; the boundary is not expanded to include the friend (line 12), and the friend's location is not added to the map (line 21).

The `friendAccess` argument allows the caller to specify a bound on the access labels of the friends who are fetched. This allows a user to plot friends stored on other social networks (modeled in the example as different stores), while preventing the user from fetching those objects if the friends' social networks are not trusted to learn about the state of the computation. Specifically, a friend's location may cause a fetch of a user object from the friend's social network. Therefore, on lines 11 and 20, before accessing the friend's location, FriendMap checks to ensure that the friend's store `friend.store` is permitted to learn information as restrictive as the `friendAccess` label. If not, then the friend is omitted from the map and the object is never accessed, preventing unsafe communication with the friend's social network.

In addition to these dynamic checks, this code requires further relationships between various labels in order to be considered secure. These relationships are demanded by the `where` clauses on lines 2–5, which are verified to be true in any method that calls `createMap`.

For example, the first clause (on lines 3–4) is concerned with information from three sources: the code itself (labeled `provider` and `MapService.provider`),[10] the fact that the method was called (bounded by `friendAccess` in the begin label), and the set of the user's friends (labeled {⊤←`user`}). These must all be able to affect the resulting map (labeled `resultL`), as well as fetches of the user's object, the map service's initial map, and the friends (with access labels {⊤→`user.store`}, {⊤→`ms`}, and `friendAccess` respectively).

Fabric requires the FriendMap developers to insert these checks. Without them, the application would fail to compile, and thus users would not be able to execute the code. Indeed, omitting any of the `where` clauses or the dynamic checks in this example could lead to exploitable information flows in the FriendMap application. Hence, these checks are not an artifact of Fabric; any secure implementation of the application would have to include similar checks to protect users' privacy. Fabric gives developers compile-time guidance about where run-time security checks are needed to achieve information security.

5. Secure mobile code

The challenge of supporting mobile code in Fabric is maintaining both strong and decentralized security guarantees while giving adversaries the power to upload and execute mobile code. Local code written in Fabric, FabIL, or Java is presumably trusted, but Fabric workers only execute mobile code if it is written in Fabric.[11] The typical workflow for compiling, linking, and loading mobile code is depicted in Figure 9.

Like any other persistent information in Fabric, mobile code is stored in persistent objects: *class objects*, shown in the middle of Figure 9. Fabric class objects should not be confused with Java class objects. A class object defines the structure and behavior of a mobile Fabric class, including the relevant security policies, and also contains references to the class's dependencies. The Fabric compiler and linker verifies and publishes new class objects by using a Fabric worker to create new objects on a designated store. This process is depicted in the "Developer" portion (the left side) of Figure 9.

Each Fabric object contains a reference to its class object. When a worker encounters an instance of a mobile class, it uses this reference to fetch the corresponding class object, as shown in the "User" portion (the right side) of Figure 9. Once it has the class object, the worker verifies the information flows within

[10] Provider labels are described in more detail in Section 5.1.

[11] It would also be consistent with the decentralized security principle for workers to download and run code written in other languages if it is signed by a node that the worker trusts.

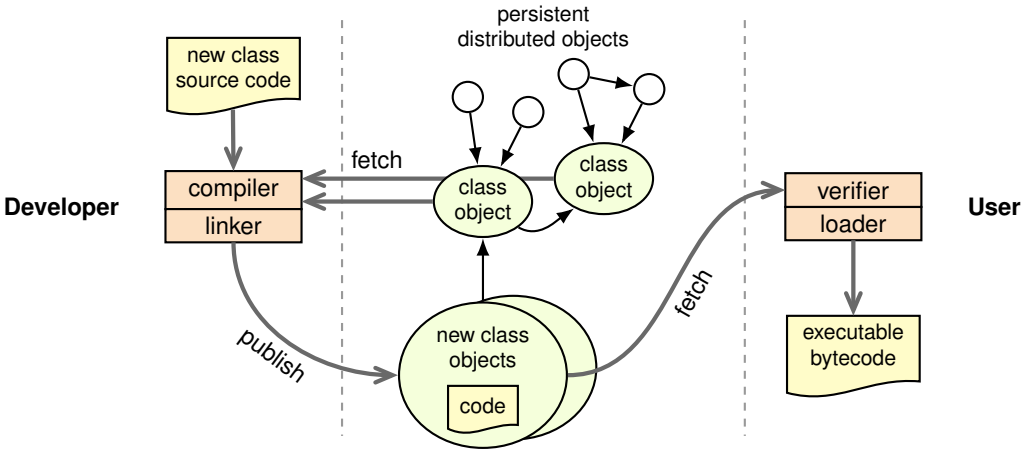


Fig. 9. Compiling, linking and loading mobile code

the class's code. Finally, it converts the class into executable bytecode, enabling the invocation of class methods.

All modern languages suffer from the problem of version conflict between program components. This occurs when two components used by a program require different versions of a third component. A common approach to avoiding such errors is to deploy each component in a package that specifies the required versions of the component's dependencies. These specifications make it possible to determine the compatibility of two components before they are loaded.

While this approach is an adequate solution for some systems, it does not address the case where a system needs to interact with multiple versions of a component. This case is particularly important for decentralized, persistent object stores like Fabric's since the lifetime of objects may span multiple releases of their dependencies, and remote hosts may be running different software versions.

Fabric's approach builds on existing approaches to component distribution by publishing linking specifications called *codebases* with all mobile code. Instead of requiring programmers to develop these by hand, the Fabric compiler generates them automatically during the publishing process. Furthermore, developers are able to interact with multiple versions or implementations of components using *explicit codebases* while keeping their namespaces isolated. These features allow developers to use user-friendly class names without worrying about namespace collisions or specific version requirements.

Importantly, name resolution and namespace isolation in Fabric are orthogonal to security enforcement, unlike in systems such as Java, JavaScript, and SPIN [39]. Many security mechanisms based on namespace isolation have proved fragile, requiring ad hoc mechanisms to prevent security vulnerabilities [**?**]. In Fabric, access to resources is restricted by information flow control rather than by limiting the ability to name those resources. Linking against high-integrity, high-authority code requires no special privilege; instead, label checking ensures the end-to-end security of linked code.

5.1. Provider-bounded checking

For local code, label checking of the sort performed in previous languages such as Jif is enough to ensure that all flows in code obey the information flow ordering. However, Jif-style label checking is not enough to stop adversary-provided code from introducing vulnerabilities. For example, Figure 10a demonstrates how integrity could be harmed if label checking worked the way it does in Jif

```
String{user←} password;
void changePassword{user←}
    (String{user←} pwd) {
  password = "weakpassword";
}
```

(a) Mobile code creating a vulnerability

```
String{user←} password;
void changePassword{user←}
    (String{user←} pwd)
    where provider ⊑ {user←} {
  password = "weakpassword";
}
```

(b) Provider-bounded mobile code

Fig. 10. Insecure and secure versions of a mobile change-password method. Confidentiality components of labels are elided.

and prior information-flow mechanisms. The method `changePassword` label-checks in Jif, because only high-integrity information appears to influence the high-integrity variable `password`: both the literal "`weakpassword`" and the method's begin label are high-integrity. The literal "`weakpassword`" is considered trusted since it is a constant, and the pc at any invocation of this method must flow to `{user←}`. However, if the adversary convinces a trusted worker node to use this code, it might be used to change the password to a string of the adversary's choosing, creating a clear security vulnerability.

Fabric eliminates such vulnerabilities by treating the code itself as information that taints the results it produces. Like other objects stored in Fabric, each class object is given an update label. We refer to this label as the class's *provider label*. Class source may explicitly refer to the provider label of any class, including its own, using the name *classname*`.provider`.[12] The provider label can appear in label annotations checked at compile time, and can also be compared to other labels at run time. The compiler makes no assumptions about the provider label during label checking. Instead, all assumptions about the code's provider are made explicit in the source code so that the same source code can be securely reused, relinked, or provided by different principals.

When label-checking code, we taint the pc label with the provider label. This makes sense because the code provider affects the statements that are executed. If the code of Figure 10a is provided by an adversary, its low-integrity provider label will effectively make the pc label low-integrity, preventing any assignments to high-integrity variables such as the password. We refer to this analysis as *provider-bounded label checking*. This approach also preserves important features of the Fabric programming model like polymorphic labels in method signatures. A more direct approach to provider checking, such as ensuring the integrity of begin labels are bounded by the provider label at load time, would require method signatures to be concrete.

Figure 10b demonstrates the use of the provider label in a `where`-clause, asserting that the provider label flows to `{user←}`. This allows the compiler to assume that the provider is sufficiently trusted by `user` to change the password. This constraint is then enforced at all call sites of `changePassword`, ensuring the password may only be updated by this code when its provider has sufficiently high integrity.

As an additional benefit, the provider label makes a new feature possible using the same mechanism: confidential code. By creating code with a high-confidentiality provider label, code publishers can safely put code into Fabric that contains sensitive information such as proprietary algorithms.

A confidentiality label on a class object prevents untrusted nodes from viewing that code directly. Provider-bounded label checking enforces a stronger notion of security, however: it prevents any data

[12] If *classname* is omitted, it defaults to the enclosing class.

affected by the code from flowing to untrusted nodes. Furthermore, it ensures (via access-label checks) that objects accessed by confidential code must be at least as high as the confidentiality of the code. These checks strictly protect the confidentiality of code by default, but providers of the code may explicitly declassify results or accesses when desired.

The provider label also enforces robust downgrading since it prevents the adversary from exploiting downgrading to affect confidentiality and integrity. A provider label with low integrity prevents the adversary from using declassification and endorsement in provided code directly; it also prevents the adversary from indirectly influencing declassification and endorsement occurring in other code not provided by the adversary.

Another curb on the misuse of downgrading operations is the requirement that code possess the *authority* to perform these operations. Code cannot weaken a confidentiality or integrity policy through declassification or endorsement, unless the principal whose policy is being weakened grants the code that authority.

Authority placed in mobile code cannot exceed the authority of the code developer. This is expressed using a check on integrity: for code that claims the authority of the principal p, Fabric ensures the condition $I(\texttt{provider}) \geqslant \{\top \leftarrow p\}$.

Fetching a mobile Fabric class on demand introduces a new kind of read channel. Therefore, the class object of a persistent object must be stored on a node that is trusted to enforce the object's access label. To satisfy this requirement without unnecessary restrictiveness, we can ensure that when an object is created on a node, its class object is stored at a suitably trusted node. Since the node storing the object itself must be such a node, the class object can be replicated onto the same node as the object if necessary. Since class objects are immutable and distinguished by their fingerprints (Section 6.5), their replication is harmless in Fabric.

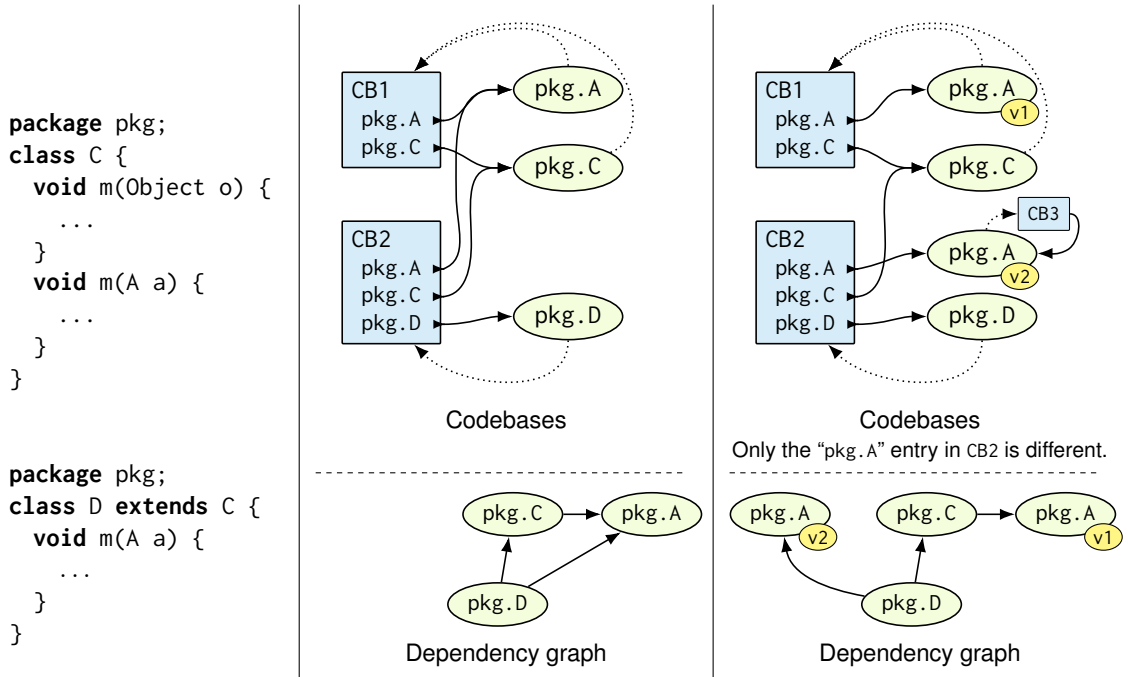
5.2. Codebases

Mobile classes are identified by their class object. However, instead of requiring programmers to insert Fabric references (i.e., URLs) to class objects directly into their source code, we allow programmers to name classes using the familiar Java naming scheme of qualified identifiers. To map class names to mobile classes, the compiler is provided with a *codebase* object, which serves a role similar to that of a Java classpath. A codebase maps from qualified class names to global, persistent references to published class objects.

Each mobile class is published with a reference to its associated *home codebase*, which the compiler uses to resolve the direct dependencies for that class. This mechanism enables independent nodes to resolve external dependencies in a uniform way without resorting to a global namespace. In contrast, Java offers no guarantee that the class linked by the JVM at run time is the same class the code was compiled against.

Because Fabric types include security policies, it is important that hosts agree on the type signature of a remote call or persistent object. Resolving names dynamically could result in errors that are difficult to diagnose. For example, a client could experience a run-time error because objects created by remote hosts resolved names differently. Furthermore, resolving names statically removes the possibility that name resolution could be used as a covert channel. Thus, mapping class names to type signatures statically is important for the security and programmability of Fabric programs.

This approach also ensures a kind of backward compatibility: new code cannot break existing instances of persistent objects. Class objects and codebases are immutable, so codebases bind class names to



(a) A mobile superclass and a separately published subclass

(b) `pkg.C` and `pkg.D` linked with the same version of `pkg.A`

(c) `pkg.C` and `pkg.D` linked with different versions of `pkg.A`

Fig. 11. Two classes linked under different scenarios

specific, fixed versions of the linked libraries. Similarly, once an object is created, its class is fixed to a specific class object. Hence, the object remains associated with the class version it was created with. New code must be backward-compatible when processing legacy objects. To "update" a legacy object to a new class version, a new object must be made. Only source-level names are resolved at compile time, so binding names to types through their codebases does not affect late binding. In fact, method dispatch can cause the fetching, compilation, and loading of code that didn't exist at compile time, similar to dynamic class loading and linking in Java.

Figure 11a shows an example of two separately published classes, `pkg.C` and a subclass, `pkg.D`. Both classes depend on a third class, `pkg.A`. The top of Figure 11b depicts the resulting codebases when `C` and `D` are linked against the same version of `A`. Solid arrows indicate the class object a particular class name resolves to, and dotted lines indicate the home codebase of a class.

Before running the code stored in the mobile class `D`, the worker invokes the Fabric compiler to verify `D`. During verification, the compiler resolves the superclass by consulting `D`'s codebase, `CB2`. The entry for "`pkg.C`" contains a reference to the specific class object used to resolve the dependency. Likewise, when verifying `C` the compiler uses `C`'s codebase (`CB1`) to resolve dependencies. This process ensures that all nodes will resolve `D`'s dependencies in the same way, including transitive dependencies (e.g., the dependencies of `C`).

Classes share the same home codebase when they are published together. In Figure 11b, `A` and `C` were published together, and therefore have a common home codebase. Class `D` has a different home codebase,

because it was published later. Because C's home codebase is different, it may contain entries not present in D's home codebase.

Rather than forcing developers to create codebases by hand, the Fabric compiler generates a codebase automatically from the classpath and sourcepath specified during publication. This feature makes the potentially complex process of linking and publishing reusable mobile components similar to compiling programs with a traditional Java compiler and linking with local libraries. Usually, classpath entries refer to the codebases of dependencies already published in Fabric, while sourcepath entries refer to local directories containing source that will be published with the new codebase.

To protect the linkage of classes resolved by a codebase, codebases have integrity labels that are at least as high as the provider labels of the classes they are published with. The compiler checks that this invariant holds at link time to prevent an adversary from exploiting codebases that violate the invariant.

5.3. Namespace consistency

Programs are often built from components that share common dependencies. Usually, these dependencies must resolve identically for all components. For instance, a dependency might define an interface through which the components interact. But in some cases, a component's use of a dependency is isolated from other components. For example, two components might require different versions of a regular expression library for manipulating strings internally. In principle, it should be safe to load both versions, since their usage is isolated. Unfortunately, identifying whether two conflicting dependencies are truly isolated from each other is difficult.

Consider the code fragments shown in Figure 11a. Imagine `pkg.D` extends a previously published class `pkg.C`. Whether the method in `D` overrides `C.m(A)` depends on how `A` is resolved. If `A` resolves to the same class used by `C`, then `D`'s method overrides `m`, otherwise it *overloads* `m`. This behavior is consistent with that of Java. Interestingly, both cases can result in fully type-checked code. Figure 11b shows the former case, in which the two codebases resolve `A` identically; Figure 11c shows the latter case, where `A` is resolved differently by `D`'s codebase, `CB2`. The bottom of the figures show the dependency graphs induced by the two scenarios. Nodes in the graph are published class objects, and each edge is the resolution of a dependency using the source node's codebase. In Figure 11c, two implementations are reachable for `pkg.A`—the program's namespace is ambiguous, or *inconsistent*.

Allowing programs with inconsistent namespaces could result in subtle and surprising behavior. For example, Fabric, like Java, dispatches calls to overloaded methods based on the static types of their arguments. Consider the code below:

```
1 pkg.A a = ... ;
2 pkg.D d = new pkg.D();
3 d.m(a); // Invokes D.m(A)
4
5 pkg.C c = d;
6 c.m(a); // Invokes C.m(Object)!
```

Let us assume that the code is compiled using `CB2` to resolve names with the dependency graph in Figure 11c. This code creates an instance `d` of class `pkg.D`, and then calls `m` on it twice, passing in a parameter of type `pkg.A` from `CB2`. How each call to `m` is dispatched depends on the type of the reference to `d`. On line 3, the reference has type `D`, so `D.m(A)` is called, as one might expect. However, on line 6, the reference has type `C`. Although `C` defines a method `m(A)`, it uses the version of `pkg.A` from `CB1`, which

```
package pkg;
codebase cb3;
class D extends C {
  void m(cb3.A a) { ... }
}
```

Fig. 12. Using an explicit codebase

is incompatible with the version from CB2 used by the actual argument. The call on line 6 is therefore statically dispatched to C.m(Object). While the author of the code might have intended this behavior, it is also likely to be unintentional.

Another reason for namespace consistency is to support *external principals* [**?**], a feature inherited from Jif that supports principals implemented with the singleton design pattern [35]. Programs mention external principals by class name, so namespace consistency ensures that when two compilation units mention the same name, they are talking about the same principal. Otherwise, information might be implicitly relabeled when it passes between compilation units.

Since errors related to inconsistent namespaces are difficult to detect and debug, we require that the static dependencies of mobile code must have a namespace that is consistent. In the dependency graph of Figure 11c, two implementations are reachable for pkg.A, so the compiler would reject the publication of pkg.D. However, if the programmer truly intends to link against two versions of pkg.A, Section 5.4 shows how this can be done with *explicit codebases*.

The consistency constraint applies to the *static* dependencies of a class and does not constrain the dynamic type of objects beyond normal type safety. At run time, a reference may point to an object whose class type is neither in the codebase of the reference's type, nor consistent with the program's namespace. There is no possibility for confusion, because our constraint ensures that dependent code only interacts with the object via an unambiguously resolved supertype.

To avoid publishing code with inconsistent namespaces, the Fabric compiler checks code at publication time for namespace consistency. At run time, the code is checked again for namespace consistency during verification, before being loaded and executed.

5.4. Explicit codebases

To support code evolution and reuse, Fabric has *explicit codebases*, which allow components to refer to multiple implementations of a dependency. An explicit codebase is an alias for a previously published codebase object. A programmer may use this alias to qualify dependencies that should be resolved through the specified codebase rather than through the home codebase of the dependent class. Explicit codebases may appear at the root of any fully qualified type name. When a name is qualified with an explicit codebase, the namespace of the specified dependency is isolated from that of the dependent class. Explicit codebases are resolved when code is published, and are exempt from namespace-consistency checks since the programmer's intention is unambiguous.

For instance, in Figure 11a, to overload C.m(A) with a different version of A, the definition of D should read as shown in Figure 12. When publishing this class, the command line associates the alias "cb3" with a Fabric reference to codebase CB3.

We expect explicit codebases to have two main uses. The most common use is to support evolving published code. Using explicit codebases, classes may provide methods or implement interfaces that

Feature	Syntax	Static checks	Section
Object persistence	`new C@s(...)`	—	3.1
Transparent data shipping	—	—	3.1
Remote calls	`f@n(...)`	$\texttt{rw} \succcurlyeq C(\ell_s)$, $\texttt{cw} \succcurlyeq I(\ell_s)$, $\texttt{rw} \succcurlyeq I(\ell_r)$, $\texttt{cw} \succcurlyeq C(\ell_r)$	3.1, 4.7
Transactions	`atomic {...}`	—	3.2
Trust ordering	$\ell \succcurlyeq \ell'$	—	4.3
"enforces" relation	$p \succcurlyeq \ell$	—	4.4
Update label ℓ_u	—	$n \succcurlyeq \ell_u$	4.4
Access label ℓ_a	$\texttt{C\{}\ell\texttt{\}@\{}\ell_a\texttt{\}}$	$pc \sqsubseteq \ell_a$, $n \succcurlyeq \ell_a$	4.6
Provider label	`C.provider`	(provider taints pc) $I(\texttt{provider}) \succcurlyeq \{\top \leftarrow p\}$	5.1
Codebases	—	(namespace consistency)	5.2, 5.3
Explicit codebases	`codebase cb;`	—	5.4

Fig. 13. Summary of features that Fabric adds to Jif

preserve compatibility with code and persistent objects from older class versions. A second use for explicit codebases is for composing software components with conflicting dependencies. If software components have conflicting dependencies that do not otherwise affect program functionality, it may be desirable to isolate the namespace of each component using an explicit codebase.

5.5. Software evolution

Instead of using dynamic linking to evolve software, Fabric relies on the inheritance and subtyping features of its object-oriented language. In Fabric, new versions of classes are stored in separate class objects, so that they can coexist and interact in the same system. If the new versions of classes are backward-compatible, then they should be subtypes of the old versions; this allows existing software that links against the old version to interact with objects using the new version. If the new version is incompatible with the old version, then it should not be a subtype: this forces software that relies on the old functionality to be updated as well. In short, there are no different versions of "the same" class: if two versions of a class are different, then Fabric considers them to be different classes.

To explain the process of software evolution, consider a possible upgrade path for the FriendMap application. Suppose that Snapp decides to extend their service by adding a `mood` field to the `com.snapp.User` class. They would do so by creating a new class, also called `com.snapp.User`. The new class would explicitly extend the old class, which means that the existing FriendMap code (which the Snapp developers have no control over) can continue to work, even with objects of the new `User` type.

At a later time, the developers of FriendMap may wish to release a new version that uses the `mood` of the user's friends to color the annotations that are placed on the map. FriendMap cannot assume that all users have the new `User` type (and thus the `mood` field). For example, the existing users Alice and Bob may not yet be upgraded to the new versions. Therefore, FriendMap must decide how it should handle old users.

One possibility would be to change the FriendMap interface so that it only accepts objects of the new `User` type. This would prevent the new FriendMap from being a subtype of the old FriendMap. Alternatively, the new FriendMap application could use dynamic downcasting to detect and use the new `mood` field, and remain backward-compatible.

The key point is that the language forces the FriendMap developers to consider the ramifications of updating their application. They may indicate whether new versions are backward-compatible by subclassing the old versions, or not.

5.6. Summary

To aid the reader in keeping track of the many language features discussed here, the table in Figure 13 summarizes the language features, syntax, and static checks that Fabric introduces on top of Jif, along with pointers to the section of the paper discussing each such feature.

6. System design

So far, we have concentrated on the Fabric programming language abstraction rather than on the system that implements it. Though some aspects of the current implementation have surfaced in this discussion, one could imagine implementing the programming abstraction in more than one way. However, any implementation of the Fabric programming abstraction must enforce the security policies expressed as labels. Enforcing these policies in a distributed system is a significant challenge, since the behavior of implementation-level mechanisms may leak information (thereby violating confidentiality), or may offer adversaries opportunities to subvert code or data (thereby violating integrity).

Fabric programs execute on the Fabric runtime system, shown in Figure 2. The Fabric runtime is an open distributed system that anyone can join by starting a new node. Each Fabric node is either a store or a worker, though a single host machine can have multiple Fabric nodes on it, often colocated in the same Java virtual machine. For example, a store typically has a colocated worker, allowing the store to invoke code at the worker with low overhead,[13] and allowing the worker to efficiently execute queries against the store.

6.1. Objects

Fabric objects are named by object identifiers (*oids*). An oid has two parts: the name of the object's store, and a 64-bit object number (*onum*) that is unique within the store. Object identifiers are global and persistent; they can be exported from Fabric by writing them as *Fabric URLs* of the form `fab://`$store$`/`$onum$. Fabric applications can also implement their own naming schemes using Fabric objects. For example, a naming scheme based on directories and path names is easy to implement using a persistent hash map.

Each object identifier is permanent in the sense that it continues to refer to the same object for the lifetime of that object, and the identifier can always be used to find the object. If an object moves to a different store, it acquires an additional oid. The original identifier still works as long as the original store keeps a forwarding pointer in a *surrogate object*.[14]

Knowing the oid of an object gives the power to name the object, but not the power to access it: *oids are not capabilities* [30]. If object names were capabilities, then knowing the name of an object

[13] Stores typically invoke Fabric code on a worker when performing dynamic authorization checks, such as when a store needs to determine whether an object's update label allows a remote worker to modify the object.

[14] Long forwarding chains of surrogate objects can reduce performance and reliability, but path compression can prevent such chains from building up [34,48,27].

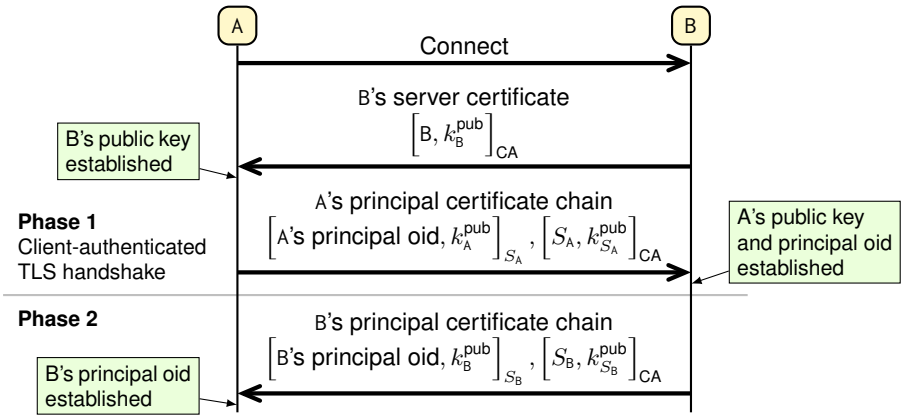


Fig. 14. Authentication protocol sequence. For simplicity, only certificate messages are shown in the TLS handshake.

would confer the power to access any object reachable from it. To prevent covert channels that might arise because adversaries can see object identifiers, object numbers are generated by a cryptographically strong pseudorandom number generator. Therefore, we assume an adversary cannot feasibly probe for the existence of a particular object, and that an oid itself conveys no information other than the name of the node that persistently stores the object.

Fabric objects can be mutable. Each object has a *current version number*, which is incremented when a transaction that updates the object is committed. The version number distinguishes current and old versions of objects. If a worker node tries to compute with out-of-date object versions, the transaction will fail on commit and will be retried with the current versions. The version number is an information channel with the same confidentiality and integrity as the fields of the object; therefore, it is protected by the same mechanisms.

6.2. Node authentication

Fabric programs identify Fabric nodes using DNS hostnames. In the runtime, each node has two additional forms of identity: a public key and a principal object. The public key is used by TLS to establish encrypted communications and to sign certificates. The principal object places the node in the principal hierarchy and is used in dynamic enforcement checks, described in Sections 4.4 and 4.7. Fabric uses X.509 certificates [42] to bind these three identities together. Certificate authorities (CAs) are therefore the roots of trust, as they are in the Web. Whether the certificates of a given CA are accepted is decided by the Fabric node receiving them.

Fabric nodes mutually authenticate when communicating over the network. Authentication establishes the remote node's principal object, so that it can be used in dynamic checks. Figure 14 shows a protocol sequence diagram for the mutual authentication of nodes `A` and `B`, which occurs in two phases. First, `A` connects to `B` and performs a *client-authenticated* TLS handshake. Every node `B` has a *server certificate*, which is an X.509 certificate signed by a CA.[15] This certificate binds `B`'s DNS hostname to `B`'s public key, similar to a TLS certificate for the Web.

Every node `A` has a *principal certificate* for performing TLS client authentication. This is an X.509 certificate that binds the oid of `A`'s principal object (`A`'s *principal oid*) to `A`'s public key, and is signed

[15] In practice, the server certificate can be part of a certificate chain rooted in a CA.

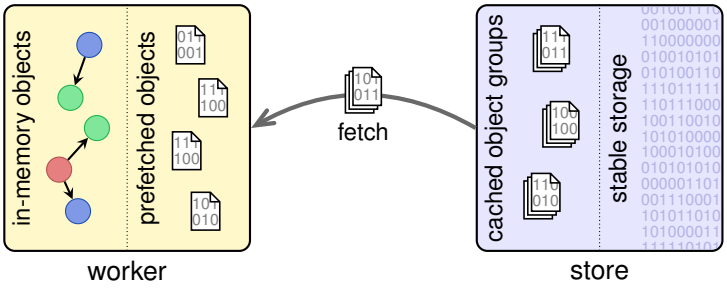


Fig. 15. Fabric hierarchy of caches

by the store S_A that hosts the principal object. When combined with S_A's server certificate, this gives a CA-rooted certificate chain for A's principal oid.

After a successful TLS handshake, A knows it has contacted the correct node, because it knows the remote node has B's private key. Similarly, B knows A's principal oid because of TLS client authentication, so A is authenticated to B.

In the second phase, B completes the mutual authentication by sending its principal certificate to A. This authenticates B if A can validate the certificate's signature, and can match the public key in the certificate with the one in B's server certificate.

6.3. *Object fetching*

When a Fabric program follows an object reference, the worker ensures that there is an in-memory copy of the object for the program to use. It does this through a hierarchy of caches, shown in Figure 15.

First, it checks to see whether a copy of the object is already present in memory. If not, the worker then consults a local cache of *prefetched objects*. These are local copies of objects that the worker has received, but not yet deserialized. The worker may have prefetched objects; when it requests an object from a store, the worker receives a group of related objects in an *object group*, amortizing both the cost of fetching and the cost of performing dynamic authorization checks at the stores.

Storage nodes dynamically construct and cache object groups based on the object graph, update labels, and dynamic access patterns. The binding between an object and its group is not permanent; the store constructs object groups as needed and discards infrequently used object groups. To improve locality, the store tries to create object groups from objects that are connected in the object graph and that have the same confidentiality policy. Having a single confidentiality policy for the entire group allows the store to treat the group uniformly with respect to confidentiality: when an authorized worker requests any member of the group, the store can safely respond with the whole group.

6.4. *Dynamic authorization*

Fabric performs three kinds of dynamic authorization checks when objects are accessed over the network. An example will illustrate these checks. Suppose Alice's worker runs a transaction that accesses and updates an object with update label ℓ_u and access label ℓ_a on Bob's store.

When Alice fetches the object,[16] the access label ℓ_a bounds how much information Bob can learn about the context of the access. However, if the reference to the object is provided by an adversary, there

[16]It is actually Alice's and Bob's nodes that are performing the actions and observations described, but for ease of exposition, we identify Alice and Bob with their respective nodes.

is no guarantee that Bob is trusted to learn that information. Therefore, before the fetch is performed, Alice dynamically checks that the store can enforce the access label: bob $\succcurlyeq \ell_a$. This check is done as part of the transaction in which the object is being accessed. If the check succeeds, Alice requests the object from Bob.

On receiving the request, Bob examines the confidentiality part of the object's update label. If Alice is trusted to read the object (alice $\succcurlyeq C(\ell_u)$), then Bob responds with the object. When Alice updates the object and commits, she sends the updated object to Bob. Before accepting the update, Bob checks that Alice is trusted with the integrity part of the object's update label: alice $\succcurlyeq I(\ell_u)$. Both of these checks are done in a fresh top-level transaction at a worker colocated with Bob.

6.5. Fingerprinting

The security of Fabric requires that trustworthy nodes agree on the types of classes. This agreement is especially important when untrusted nodes can provide class objects. Suppose Alice's worker wishes to make a remote call to Bob's worker to execute an attacker-provided method `harmlessMethod`. The attacker can try to provide two different implementations of the method to the two nodes. Suppose Alice sees the code on the left, whereas Bob receives the implementation on the right:[16]

```
// As seen by Alice
void harmlessMethod{alice→bob} () {
}
```

```
// As seen by Bob
void harmlessMethod{alice→⊥} () {
  publicData = true;
}
```

Both of these methods type-check. Alice is willing to make this remote call in a context that would reveal confidential data (alice→bob), because the begin label of `harmlessMethod` prevents it from having any public side effects. On the other hand, Bob is willing to execute the method even though it has a public side effect, because its begin label requires it to be called in a context that does not reveal any sensitive information.

However, if the two implementations can be combined, the type mismatch allows the attacker to trick Bob into revealing confidential information. It is not enough that the methods both type-check in isolation; Alice and Bob must *agree* on the types.

To ensure that the caller and the receiver of a remote call agree on the types appearing in remotely called methods, a *fingerprint* [11] is sent with each remote call request. The receiver checks that the invoked method has a matching fingerprint. The fingerprint is computed as a cryptographic hash over the entire source code of the method's class, including the source code of all superclasses.

Fingerprinting ensures more type agreement than is strictly necessary, but there is no harm in it, since the same fingerprint has other uses. For instance, a similar vulnerability exists if an attacker can cause nodes to disagree about the labels on the object's fields by changing the class associated with the object. To prevent this attack, each object stores its class fingerprint along with the pointer to its class object. The fingerprint is checked against the class actually loaded to verify that the class accurately describes the object, including security policies on its fields.

6.6. Transaction management

Fabric uses a mix of pessimistic and optimistic concurrency control. To coordinate threads running on the same worker, Fabric uses pessimistic concurrency control, in which threads acquire locks on objects.

However, in the distributed setting, Fabric is optimistic: worker nodes compute on cached copies of objects that may be out of date, while a distributed two-phase commit protocol [37] ensures consistency at commit time.

Within a single worker, multiple threads may execute concurrently. Every thread in the worker has a transaction manager that maintains the state of the thread's transaction, and each object in the worker's cache has a reader–writer lock. During computation, the transaction manager logs the version numbers of objects that are read or written, and the identities of the objects that are created. When a thread reads or writes an object, the transaction manager acquires a read or write lock for the object. The thread blocks if the lock would conflict with another lock held by a different thread.

Updates to objects in a transaction are logged in an undo log: upon the first write to each object during a transaction, the transaction manager logs the prior state of the object in an *object history*. This is used to restore the object's state in case the transaction fails.

Transaction logs and object histories are hierarchical because transactions can be nested. When a local subtransaction is created, it inherits the locks held by its parent. When the subtransaction commits, its log is merged with the parent transaction log, and its locks are transferred to the parent transaction. If the subtransaction aborts, it discards its log, relinquishes the locks it has acquired, and restores the state of the objects it has modified.

To reduce logging overhead, the copy of each object at a worker has a *reader stamp*, which is a reference to the last transaction that read the object. No logging needs to be done for a read access if the current transaction matches the reader stamp. Similarly, each object has a *writer stamp* for the last transaction that modified the object, and no logging is needed if the current transaction matches the writer stamp. Obtaining the write lock clears the reader stamp.

When a worker commits a top-level transaction, it initiates a two-phase commit protocol with the stores for the objects accessed during the transaction. The information sent to each store includes the version numbers of objects accessed during the transaction and the new data for written objects. For security, the store performs the authorization checks described in Section 6.4 to ensure that the worker is trusted to modify the written objects.

The store also checks that its authoritative version numbers match the version numbers reported by the worker. A mismatch means that another worker has updated an object that was accessed during the transaction. In this case, the transaction fails, and the store informs the worker which objects involved in the transaction were out of date and provides fresh copies of the objects. The worker then replaces the stale objects in its cache before retrying the transaction.

Transactions can also fail if the state of the objects read by the transaction are not mutually consistent. Inconsistent objects might break invariants that the application code relies upon, causing errors in the execution of the application. Errors that lead to incorrectly computed results are not a problem because they will be detected and rolled back at commit time. Errors that cause exceptions are also possible, but as discussed earlier, exceptions also cause transaction failure and rollback. Finally, an application's computation might diverge rather than terminate. Fabric handles divergence by retrying transactions that are running too long. On retry, the transaction is given more time in case it is genuinely a long-running transaction. By geometrically growing the retry timeout, the expected running time of long-running transactions is inflated by only a constant factor.

The use of reader–writer locks at workers means deadlocks are possible. Fabric implements local deadlock detection and aborts transactions detected to be deadlocked. Though distributed deadlocks may occur in Fabric, there are well-known techniques (e.g., edge chasing [20]) for detecting and avoiding

them in a non-federated context. Transactions stuck in distributed deadlocks eventually time out and are retried. We leave to future work the design and implementation of secure, federated deadlock detection.

6.7. Security cache

At run time, Fabric applications perform acts-for tests and label comparison tests that are needed for security, but that add performance overhead. Fabric reduces this overhead by memoizing the results of these tests in a *security cache*. The cache has separate partitions for positive and negative results of acts-for tests and label comparisons, similar to the cache used by SIF [24].

Soundness is maintained in two ways. First, the cache is adjusted when the application changes the principal hierarchy. The negative partition is cleared when a principal delegation is added. Similarly, when a delegation is revoked, any positive-cache entries that depend on that delegation are removed.

Second, the security cache is tied to the transaction manager, so concurrent transactions are isolated. Each transaction has its own security cache, and because transactions can be nested, the security cache is hierarchical. When a subtransaction is created, it inherits the cache entries from its parent. When the subtransaction commits, its cache is merged with the parent cache; if the subtransaction aborts, its cache is discarded. Thus, the security cache of the parent transaction is isolated from changes to the principal hierarchy made within an aborted subtransaction that may have computed trust using inconsistent data.

6.8. Writer maps

Remote calls cause transactions to be distributed across multiple workers. Several challenges arise when objects are shared and updated by the workers in a distributed transaction. For consistency, workers need to compute on the latest version of a shared object as it is updated. For performance, workers should be able to locally cache objects that are shared but not updated. For security, a worker should only learn about updates to an object if the worker is trusted with the object's confidentiality.

Fabric addresses these challenges with *writer maps*, a cryptographic data structure that allows workers to efficiently check for updates to cached objects, while protecting the confidentiality of those updates. Every transaction has a writer map that associates each updated object with its *writer*, the worker that last wrote to the object during the transaction. An object's writer, therefore, stores the definitive copy of the object for the transaction.

The writer map is threaded through the control flow of the distributed computation: it is included in every remote-call request, and is returned with the result of the call. During computation, when a transaction on a worker w accesses an object, the transaction manager checks the writer map. If a writer w' is found, then the worker w verifies that w' enforces the object's integrity before fetching the latest version of the object from w'.

If the object is being written, one of two things happen. First, if w can enforce the access label ℓ_a on the object ($w \succcurlyeq \ell_a$), then w' is notified to relinquish its role as writer. The notifying worker w becomes the new writer, and this change is recorded in the writer map. Notification of the old writer w' is not a covert channel, because the program counter label pc of the write must flow to the object's update label ℓ_u, which the old writer is already trusted to read: $pc \sqsubseteq \ell_u \sqsubseteq \{\top \rightarrow w'\}$. Otherwise, if w does not enforce the access label ($w \not\succcurlyeq \ell_a$), then w buffers its writes to the object, and furnishes its updates to w' on the next remote-call transfer, leaving w' as the writer.[17]

[17]The worker colocated on the object's store is the writer if none of the workers in the transaction can enforce the object's access label.

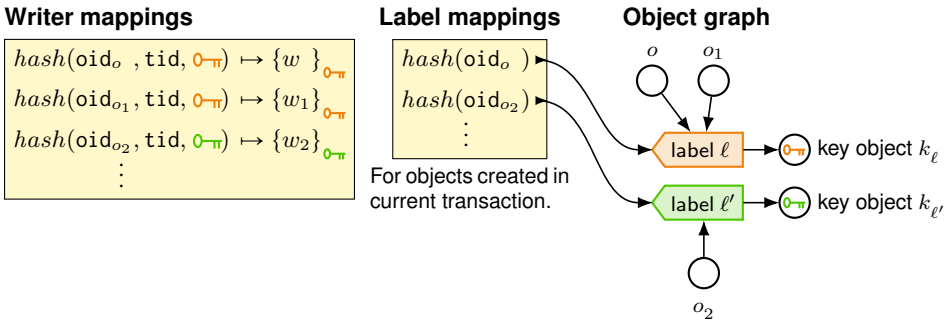


Fig. 16. Writer maps

Writer maps cryptographically associate the oid of each updated object with the object's writer, as shown in Figure 16. When a worker w updates an object o, it adds a *writer mapping*, shown on the left side of the figure. Writer mappings have the form $hash(\texttt{oid}_o, \texttt{tid}, \texttt{key}) \mapsto \{w\}_{\texttt{key}}$. The $hash$ is a cryptographic hash function (and therefore collision resistant); $\texttt{oid}_o$ is the oid of the updated object o; `tid` is the identifier for the top-level transaction;[18] and `key` is a symmetric encryption key associated with the object, under which the value w is encrypted.

A worker in possession of an object's encryption key can use the writer mapping to learn whether there is an entry in the writer map for that object, and to determine which node is currently the object's writer. Workers lacking the key cannot exploit these mappings, because the hash cannot be computed without the key. Nor can they watch for the appearance of the same mapping across multiple transactions, because the top-level transaction id is included in the hash.

Encryption keys are stored in *key objects*, shown in the object graph on the right side of Figure 16. Key objects are referenced from the update labels of objects. Each *instance* ℓ of an update label has a different key object k_ℓ, created at the same time as ℓ. The update label of k_ℓ is ℓ itself, so objects have the same confidentiality and integrity as their key objects. Different instances of a label have different key objects. However, the cost of obtaining a key object is amortized, because objects with the same update label instance have the same key object.

A worker can obtain the key object for an object o by first obtaining the update label ℓ of o, and then following the reference in ℓ. The update label ℓ can be obtained in one of three ways. First, every object has an immutable reference to its update label, so if the worker has o cached, it can simply follow this reference to obtain the update label. This works even when the cache is out of date because the reference is immutable. Second, if o is not cached, but was created in a previous transaction, the worker can obtain a copy of o from its store and then follow the reference in o to obtain the update label.

Third, if o is not cached and was created in the current transaction, then it has not yet been written to a store. To handle this, the writer map contains *label mappings*, shown in the middle of Figure 16. The creation of a new object o with oid $\texttt{oid}_o$ adds a mapping from $hash(\texttt{oid}_o)$ to the oid of o's update label. Label mappings allow a worker to find the encryption key for newly created objects and then to check for a writer mapping.

To reduce the overhead of checking the writer map, each worker in the computation keeps a local version number for the writer map, which is incremented when the worker receives new writer-map information from an incoming remote-call request or a remote-call result. Each cached object at a worker

[18] Transaction identifiers are generated by a cryptographically strong pseudorandom number generator.

has a local *writer-map stamp*, which records the version number of the writer map used during the previous access. No fetch needs to be done if the current writer-map version number matches the writer-map stamp.

The writer map is an append-only structure, so if an untrusted worker fails to maintain a mapping, it can be restored. The size of the writer map is a side channel, but the capacity of this channel is bounded by always padding out the number of writer map entries added by each worker to the next largest power of 2, introducing dummy entries containing random data as needed. Therefore, a worker that modifies n objects leaks $O(\lg \lg n)$ bits of information at each control transfer, and a computation modifying n objects across c control transfers leaks $O(c \lg \lg n)$ bits. This leakage bound for a distributed computation is conservative but should be small in practice, since $\lg \lg n$ grows slowly and we expect the number of control transfers within a transaction to be small in typical use.

6.9. Distributed transactions

The management of a distributed transaction is itself distributed across the workers participating in the transaction. Each worker maintains transaction logs for the top-level transactions it is involved in. For security, workers are responsible for logging their own actions, because they may contain confidential information that other workers should not observe. For example, in the code below, the existence of `a` or `b` in the transaction log can reveal the value of `secret`.

```
int x;
if (secret) x = a.f;
else x = b.f;
```

Figure 17 illustrates the log structures that might result during a distributed transaction involving three workers, `A`, `B`, and `C`. Each transaction, including nested transactions, is identified by a randomly generated transaction id (tid). For clarity of presentation, however, the figure uses sequential tids. Each remote-call request includes the tids for the call's entire transactional context. This allows the receiving worker to synchronize its transaction state with that of the calling worker.

In Figure 17a, a transaction (`tid=01`) starts on worker `A`, then calls a method on worker `B`, which starts a nested subtransaction (`tid=02`) there. Because the request from worker `A` includes the context `ctxt=01`, worker `B` knows that `tid=02` occurs within `tid=01`. The method then calls to worker `C`, starting another subtransaction (`tid=03`), which finally calls back to worker `B`, starting subtransaction `tid=04`. Conceptually, all the transaction logs together form a single log that is distributed among the participating workers, as shown at the bottom of the figure.

When worker `B` returns to worker `C`, it commits `tid=04`, resulting in the state shown in Figure 17b. The procedure for committing a subtransaction with a distributed transaction log is the same as for a transaction with a non-distributed log. To commit `tid=04`, worker `B` merges its portion of the log for `tid=04` with that of `tid=03`. Though worker `C` also has a portion of the log for `tid=03`, the two parts are kept separate.

Figure 17c shows the state of the distributed transaction log after worker `C` returns to worker `B`, and `tid=03` has committed. Before returning, worker `C` commits its portion of `tid=03`, so it merges its log for `tid=03` with that of `tid=02`.

When control returns from a remote call, the worker always commits up to the context in which the remote call occurred. Therefore, when worker `B` receives control, it also commits its portion of `tid=03`

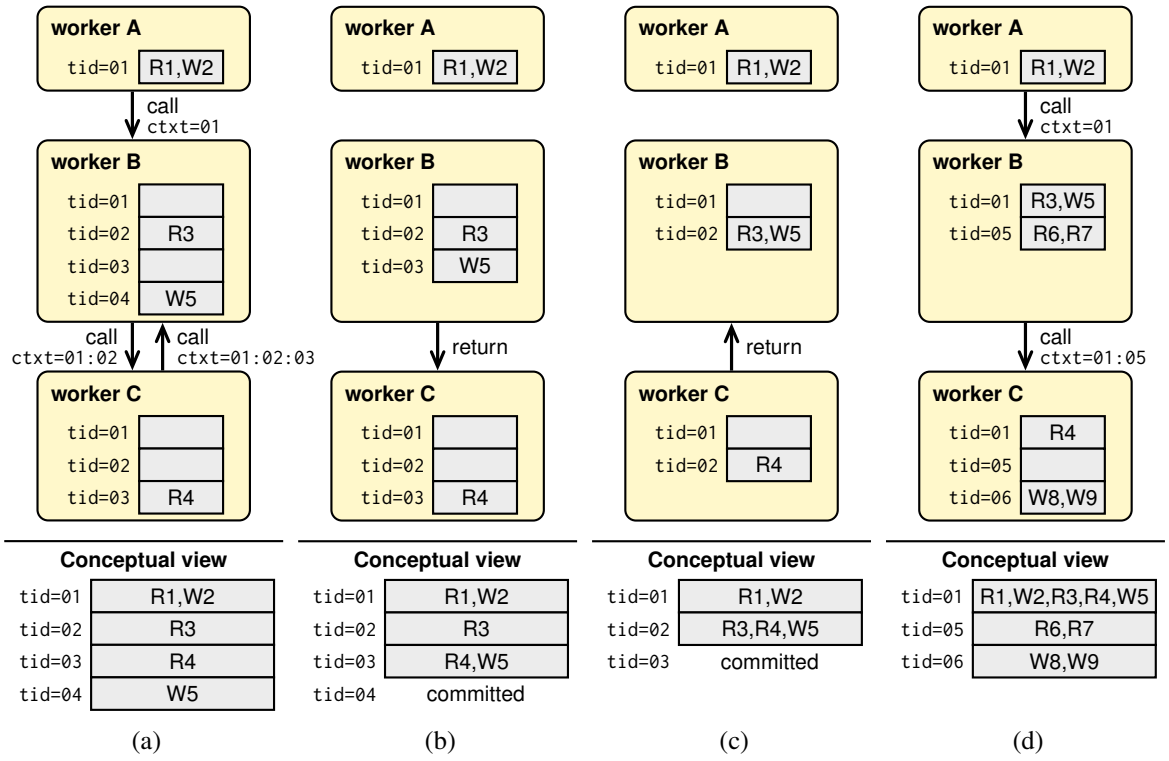


Fig. 17. Distributed transaction logs

so it can continue working within `tid=02`. This merges worker `B`'s portion of `tid=03` (which includes entries from `tid=04`) with `tid=02`.

The return from worker `B` to worker `A` is elided. The diagram looks like Figure 17c, except worker `B` has merged `tid=02` into `tid=01`. Although a similar merge does not happen at worker `C`, this is not a problem, because the merge will occur when the top-level transaction commits, or when worker `C` receives control again. Figure 17d shows the latter case. Worker `A` calls worker `B` again, which starts `tid=05` and calls worker `C`, starting `tid=06`.

When a worker receives a remote call, it compares its transactional context with the one it receives. The worker synchronizes its transaction log by committing up to the most recent common ancestor of the two contexts, and starting any transactions it has missed. Therefore, when worker `C` receives the remote call, it compares its transactional context (`01:02`) with the one it receives (`01:05`). It commits up to the most recent common ancestor, `tid=01`, and starts the transaction it has missed, `tid=05`, before starting `tid=06` for the incoming call.

When the top-level transaction commits, the three workers participate in a hierarchical commit protocol to communicate with the stores of the objects they have accessed, using their respective parts of the logs.

6.10. Hierarchical commits

A transaction may span worker nodes that do not trust each other, creating both integrity and confidentiality concerns. An untrusted node cannot be relied to commit its part of a transaction correctly. More subtly, an insecure commit protocol might cause an untrusted node to learn information it should not.

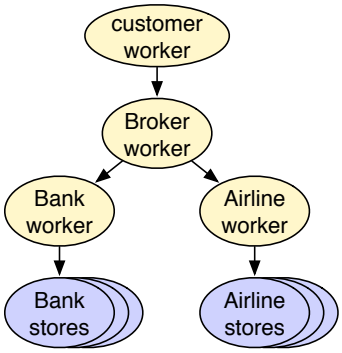


Fig. 18. A hierarchical, distributed transaction

For instance, simply learning the identities of other nodes that participated in a transaction may reveal sensitive information. Fabric's *hierarchical two-phase commit* protocol addresses these problems.

Consider the transaction shown in Figure 18, in which a customer buys an air ticket. The bank and the airline do not necessarily trust each other; nor do they trust the customer purchasing the ticket. Therefore some computation is run on workers managed respectively by the bank and the airline. When the transaction is to be committed, some updates to persistent objects are recorded on these different workers.

Because the airline and the bank do not trust the customer, their workers will reject remote calls from the customer—the customer's worker lacks sufficient integrity. Therefore, this scenario requires the customer to find a trusted third party. As shown in the figure, a third-party broker can receive requests from the customer, and then invoke operations on the bank and airline. Because the broker runs at a higher integrity level than the customer that calls it, Fabric's endorsement mechanism must be used to boost integrity. This rule is needed because the security policy of the broker says that anyone is allowed to make requests of the broker. It is the responsibility of the broker to sanitize and check the customer request before explicitly endorsing it and proceeding with the transaction.

The commit protocol is a hierarchical version of the usual two-phase commit protocol. The first phase begins with the worker that started the top-level transaction. It initiates the commit by contacting all the stores for whose objects it is the current writer in the writer map, and all the other workers to which it has issued remote calls. These other workers then recursively do the same, so the first phase of the protocol constructs a *commit tree*, a spanning tree of the transaction's *remote-call graph*. This process allows all the stores involved in a transaction to be informed about the transaction commit, without relying on untrusted workers to choose which nodes to contact and without revealing to workers which other nodes are involved in the transaction lower down in the commit tree.

The two-phase commit protocol then proceeds as usual, except that messages are passed up and down the commit tree rather than directly between a single coordinator and the stores. The first phase of the protocol not only constructs the commit tree but also causes each participating store to validate the transaction by checking permissions and comparing version numbers. Each store that successfully validates the transaction prepares to commit it. The second phase of the protocol informs all participants whether the prepared transaction should be committed or aborted.

Of course, a worker in this tree could be compromised and fail to correctly carry out the protocol, causing some stores to be updated in a way that is inconsistent with other stores. However, this worker could already have introduced this inconsistency by simply failing to update some objects or by failing

to issue some remote method calls. In our example above, the broker could cause payment to be rendered without a ticket being issued, but only by violating the trust that was placed in it by the bank and airline. The customer's power over the transaction is merely to prevent it from happening at all, which is not a security violation.

Once a transaction is prepared in the first phase of the two-phase commit, it is important for the availability of the objects involved that the transaction is committed quickly. The transaction coordinator should remain available, and if it fails after the first phase of the transaction, it must recover rapidly. An unavailable transaction coordinator could become an availability problem for Fabric, and the availability of the coordinator is therefore a trust assumption. To prevent denial-of-service attacks, prepared transactions are timed out and aborted if the coordinator is unresponsive. In the example given, the broker can cause inconsistent commits by permanently failing after telling only the airline to commit, in which case the bank will abort its part of the transaction. This failure is considered a violation of trust, but in keeping with the security principles of Fabric, the failing coordinator can only affect the consistency of objects whose integrity it is trusted to enforce. This design weakens Fabric's consistency guarantees in a circumscribed way, in exchange for stronger availability guarantees.

7. Implementation

Fabric is implemented using a mixture of Java, FabIL, and Fabric code. Altogether, the Fabric compiler and runtime system contain 45k lines of code.[19] Ports of earlier Java and Jif libraries contain an additional 17k lines of code.

A common base of 11k lines of code supports the worker and store. The worker is written in another 4.7k lines of Java code and 3.7k lines of FabIL code, and the store is an extra 2.6k lines of Java code.

The Fabric compiler is implemented as a source-to-source translation from the Fabric language to Java, using the Polyglot extensible compiler framework [70]. Fabric is translated to Java by way of FabIL. Together, the Fabric and FabIL compilers are 21k lines of code. The FabIL compiler is an extension of the Polyglot base compiler (`jlc`), while the Fabric compiler extends the Jif 3.5 compiler [**?**], itself a 37k-line extension of `jlc`. Provider-bounded label checking is implemented as part of Jif and is inherited by Fabric. The implementation of this feature and other changes needed for Fabric led to a 12k-line change to the Jif compiler.

We ported some Java and Jif libraries, both to support the runtime system and to aid application development. Partial ports of the GNU Classpath implementation of the `java.util` package resulted in 6.7k lines of FabIL code and 3.1k lines of Fabric code. The Servlets with Information Flow library (SIF) [24] provides an API for building web applications, and was originally written in Jif. Its port to Fabric contains 5.4k lines of Java, FabIL, and Fabric code. Fabric developers can use SIF to develop a familiar web-servlet front end to a distributed Fabric application, offering a new interface between Fabric and traditional web browsers and applications. SIF is built on top of the standard Java J2EE library for implementing web servlets, but raises the level of abstraction provided by that library to allow web applications to construct server-side DOM trees. The nodes in these DOM trees contain explicit information flow policies regarding both outputs sent to the browser and inputs received from it, allowing information flow to be tracked statically, unlike the fully dynamic approach taken in some later work [**?**]. SIF also

[19] In this paper, we exclude comments and whitespace when counting lines of code.

injects these policies into the generated web pages and supports automatically reflecting the sensitivity of web page inputs and outputs at the browser.

Implementing Fabric in Java has the advantage that it supports integration with and porting of legacy Java applications, and access to functionality available in Java libraries. However, it limits control over memory layout and prevents the use of some useful implementation techniques. In an ideal implementation, the virtual machine and JIT would be extended to support Fabric directly. For example, the worker cache is implemented with Java's `SoftReference` feature, and could otherwise be implemented with fewer indirections. We leave VM extensions to future work.

7.1. Store

The store implementation uses Berkeley DB [71] as a backing store in a simple way: each object is entered individually with its onum as its key and its serialized representation as the corresponding value. The performance of this simple approach is reasonable for the applications we have studied, probably because of the use of aggressive caching elsewhere in the system: recall that stores cache both object groups and object versions in memory, and workers cache objects across transactions. For write-intensive workloads, object clustering in the backing store might improve performance, but we leave this to future work.

It is important for performance to keep the representation of an object at a store and on the wire compact. Therefore, references in an object are stored as onums rather than as full oids. A reference to an object located at a different store is stored as an onum that refers to a surrogate object containing a forwarding pointer. This representation is compact if most references are to an object in the same store.

7.2. Class loading

Workers load and run mobile code using a custom Java class loader that we implemented in 79 lines of FabIL code. When the JVM requests a new class, the class loader fetches the corresponding Fabric `FClass` object, which contains the class's source code and references to the home codebase. To specify which codebase should be used, the Fabric compiler mangles class names mentioned in mobile code to include their home codebases.

After fetching a class object, the custom loader invokes the Fabric compiler on the source code to verify the class and generate bytecode. Important context information, such as the worker's principal, the run-time provider label of the code, and the codebase, are also passed to the compiler. The resulting bytecode is cached locally in memory and on disk, so that compilation can be reused. The custom loader then reads the bytecode from the cache and uses the Java class loader API to load it into the JVM. For bootstrapping purposes, some system classes are treated specially. They are loaded from bytecode on disk, in much the same way as by the default Java classloader.

7.3. Memory management

Workers cache persistent objects across transactions, which means that a mechanism is needed to evict objects from cache when they are no longer needed and the memory they occupy is needed for other purposes. The current implementation uses Java's `SoftReference` feature to evict objects that aren't needed by active transactions. References to an object via a `SoftReference` do not prevent garbage collection of the object. However, using `SoftReference` does introduce an extra indirection when following each reference between Fabric objects. With more direct control over code generation and the run-time platform,

this indirection could be removed, likely improving the performance of worker computation. However, we leave this to future work.

Note that the eviction of an object is not observable at the level of the Fabric language, so this eviction mechanism does not create a side channel between different threads at a Fabric worker.

7.4. Unimplemented features

Most of the Fabric design described in this paper has been implemented in the currently available prototype. A few features are absent, though no real difficulties are foreseen in implementing them, and their absence should not have any significant effect on the results reported here. Distributed deadlock detection via edge chasing [20], timeout-based abort of possibly divergent computations, timeout-based abort of prepared transactions for availability, class-object replication, and path compression for forwarding chains of surrogate objects [34,48,27] are unimplemented. Full support of run-time enforcement of access labels in multi-worker transactions is incomplete. Workers also do not currently verify that the writers obtained from the writer map can enforce the objects' integrity (Section 6.8). When an object is fetched from a remote node, a dynamic check is done to ensure that its class is a subtype of the expected type of the reference to the object. Currently this check does not consider parameters of parameterized types. Including these parameters is possible and conceptually straightforward but, in the current architecture, would require a major change to FabIL to better support parameterized types. The check comparing the remote node with the access label, described in Section 6.4, is also unimplemented.

Fabric does not currently support garbage collection for persistent objects. Garbage collection has security implications for referential integrity, studied in [**?**]. Designing a secure, distributed garbage-collection protocol for the federated setting remains an open problem.

8. Evaluation: Expressiveness

To explore the expressiveness of the Fabric language, we implemented various programs, including the FriendMap example of Section 2 as well as an example in which agents bid on airline tickets. These applications work correctly on our prototype implementation, demonstrating that it is possible to build interesting code under the restriction imposed by provider-bounded information flow control. Further, codebases enable incremental development of both of these examples.

8.1. FriendMap example

The prototype of Section 2's FriendMap example contains roughly 1,800 lines of Fabric code, about 44 of which implement the extended versions of FriendMap and Snapp described in Section 5.5. We added a second version of the Snapp codebase that introduces a `mood` field to `User` objects. The version 2 classes use the explicit codebase feature to refer to the version 1 classes, and the `User` class in version 2 extends the `User` class in version 1.

We also extended the FriendMap application to make use of this added functionality. FriendMap version 2 extends FriendMap version 1, and overrides the implementation of the `addPin` method to color the added pin to represent the user's mood. Because version 2 is a backward-compatible extension of version 1, it must be able to handle version 1 `User` objects that have no moods. The implementation uses explicit codebases to perform dynamic type checks, and falls back to version 1 behavior if version 1 users are encountered.

```
interface UserAgent[label L] {
  int{L} choice{L} (Offer[L]{L} offer1, Offer[L]{L} offer2);
}

interface Agent[principal A, label L] {
  void prepareForAuction{A→;A←}();
  Offer[L]{L} makeOffer{L} (UserAgent[L]{L} userChoice, Offer[L]{L} bestOffer);
  ...
}
```

Fig. 19. Interfaces provided by `Broker`

8.2. BiddingAgent example

In our second major example, a user supplies an agent for choosing between ticket offers made by two different airlines. The choice may depend on factors confidential to the user, such as preferred price or expected service level. Airlines, in turn, supply agents that compete for the best offer to provide to the user, while maximizing profit. This example comprises about 580 lines of code.

Four parties participate: a trusted broker, two airlines, and the user. They are represented by Fabric principals `broker`, `airlineA`, `airlineB`, and `user`. The broker is trusted by others: `broker` $\succcurlyeq$ `airlineA`, `broker` $\succcurlyeq$ `airlineB`, and `broker` $\succcurlyeq$ `user`. No other trust relationships are assumed. Every principal is associated with a Fabric store.

To facilitate interaction of different mobile agents, the broker publishes interfaces, illustrated in Figure 19, for the airlines' and user's agents. The interfaces use principal and label parameterization, a Fabric language feature (inherited from Jif) that facilitates modular development and genericity. Interface `UserAgent` has a label parameter `L` that corresponds to the security level of the offers that it chooses from. The `choice` function returns `-1` if the first offer is preferred, `1` if the second offer is preferred, and `0` if offers are equally preferred.

Interface `Agent` for airline agents has two parameters: `A` for the airline principal and `L` for the label of the offers. Two noteworthy methods here are `prepareForAuction` and `makeOffer`. Method `prepareForAuction` may be called before bidding starts. The begin label of this method, `{A→;A←}`, permits airline `A` to observe calls to this method. This allows airline agents to fetch new information from their airlines, such as seat availability or the current lowest prices.

Method `makeOffer` is called during the bidding phase to generate a new offer to the user's agent. The signature of this method demonstrates the key feature of our mobile-code framework: the user's agent is passed in as the method argument `userChoice`, and can be called internally by the airline agent. Similarly, the current best offer is passed as another argument, allowing the agent to find an offer better than the current best according to the user—while still trying to maximize profit. The enforcement of information-flow policies ensures that no confidential information (such as the user's maximum price or offers from competing airlines) flows from the agents to the principals that provided them, despite the fact that these agents process this sensitive information directly.

Figure 20 shows how mobile agents are initialized. Line 1 declares a label `auction` at which offers are produced. The confidentiality component of this label, `{broker→}`, records that an offer may only be read by the broker; the integrity component of this label, `{user←;airlineA←;airlineB←}`, records that the choice of an offer may be influenced by all three principals (the user and both airlines). Lines 2–4

```
1 label{broker←} auction = new label{broker→;user←;airlineA←;airlineB←}
2 Agent[airlineA,auction] agentA = a.getAgent(auction);
3 Agent[airlineB,auction] agentB = b.getAgent(auction);
4 UserAgent[auction] userAgent = u.getAgent(auction);
```

Fig. 20. Initializing airline and user agents

initialize the airline and user agents, and illustrate how principal and label parameters are provided. To inform the user of the auction result, the winning agent and offer need to be declassified. This requires the authority of the broker; for example, the broker code to declassify the winner looks as follows:

```
declassify (
  endorse (winner, {auction} to {broker→;broker←})
  to {broker→user;broker←})
```

Before declassifying the value of `winner`, this code endorses it to integrity `{broker←}`. Without this endorsement, the declassification would not be robust, because a potentially untrusted principal `user` can influence what confidential information they learn. Direct endorsement ensures that the programmer has explicitly acknowledged this influence. Declassification of the winning offer is justified similarly.

9. Evaluation: Performance

High-level programming abstractions can come at a cost that interferes with building real applications. Fabric imposes several new overheads in various parts of the system: for example, remote calls require dynamic label checks and access to local objects requires transactional logging. On the other hand, because Fabric integrates some system layers more tightly than conventional system architectures do, it also has the potential to improve overall application performance.

We evaluated the performance of the Fabric system using four applications. First, we ported Cornell's deployed Course Management System (CMS) [15] to Fabric, to evaluate performance on real-world workloads. Second, we implemented the multiuser OO7 benchmark [18] to evaluate system scalability. These first two applications are written in FabIL, and evaluate Fabric without using mobile code. Although FabIL does not enforce information flow security, its applications are transactional and subject to dynamic authorization checks at the stores, and therefore incur much of the same overhead of transaction logging, label creation, and dynamic label checks as Fabric programs do.

Support for mobile code affects the performance of the system in two ways. First, additional work is required to dynamically load and analyze new code. Second, linking with remote classes imposes some execution overhead. To evaluate these effects, we additionally ran the FriendMap and BiddingAgent programs. Measurements from these four applications suggest that run-time performance is acceptable for many uses.

9.1. Course Management System

To compare the performance of Fabric to current standard alternatives, we ported a portion of the CMS course management system to FabIL. The production version of CMS is a 38k-line Java web

application, backed by a conventional SQL database. It has been used at Cornell University to manage course assignments and grading since 2005; at present, it is used by more than 50 courses and more than 4,000 students.

CMS has a model–view–controller architecture. In the production version, the model is implemented with Enterprise JavaBeans (EJB) using Bean-Managed Persistence. For performance, hand-written SQL queries are used to implement lookup and update methods, while generated code manages object caches and database connections. The model contains 35 Bean classes encapsulating students, assignments, courses, and other abstractions. Each Bean class corresponds to a table in a SQL database, and EJB maps instances of these classes to rows of these tables. The view is implemented using Java Server Pages.

EJB programs access persistent Bean instances using an interface based on Java Collections. Porting CMS to FabIL involved replacing this interface with a FabIL collections library based on a port of GNU Classpath. We ported the entire data schema and partially implemented the query functionality of the model. By replacing complex SQL queries with object-oriented code using loops and standard data structures, we were able to greatly simplify the model code; fully porting five of the Bean classes reduced 3,100 lines of code to 740 lines, while the view and controller remained mostly unchanged.

To guide our port, we used the trace from the production CMS server described below. This ensured our port focused on the most important application features: of the 101 types of user requests in the production application, the 11 most popular were fully ported, covering 93% of the requests captured in the trace. Many of the ported request types contain significant computation and data usage. For instance, the "course" request displays an overview of all relevant course information, including announcements, outstanding assignments and their due dates, as well as graded assignments and their grade distributions. The port was completed in less than two months by an undergraduate initially unfamiliar with Fabric. It contains 18k lines of code. These results suggest that porting web applications to FabIL is not difficult and results in shorter, simpler code.

A complete port of CMS to Fabric would have the benefit of federated, secure sharing of CMS data across different administrative domains, such as different universities, assuming that information is assigned labels in a fine-grained way. It would also permit secure access to CMS data from applications other than CMS. We leave this to future work.

Workload We obtained a trace from Cornell's production CMS server from three weeks in 2013, a period that encompassed multiple submission deadlines for several courses. To drive our performance evaluation, we took 10 common request types from the trace. Each transaction in the trace is a complete user request including generation of an HTML web page, so most request types access many objects. Using JMeter [46] as a workload generator, we sampled the traces, transforming query parameters as necessary to map to objects in our test database with a custom JMeter plugin.

9.2. CMS comparison baseline: Hibernate/HSQLDB

To provide a credible baseline for performance comparisons, we ported our FabIL implementation of CMS to the industry-standard Java Persistence API (JPA) [13], an API for object–relational mapping (ORM) that, like Fabric, offers a transactional programming abstraction with persistent objects. The JPA programming model lacks security policies or security enforcement, and is roughly similar to that of FabIL, though its persistence is not orthogonal and explicit calls to the transaction manager must be inserted to ensure persistence and consistency. At a high level, the JPA code for CMS is very similar to the FabIL code, using essentially the same schema and the same algorithms.

We evaluated performance of this baseline implementation using the widely used Hibernate implementation of JPA 2, running on HyperSQL (HSQLDB), a popular in-memory database. We ran HSQLDB in "`READ COMMITTED`" mode, a weaker level of consistency than the strict serializability provided by Fabric. We refer to this configuration as JPA.

9.3. Multiuser OO7 benchmark

The OO7 benchmark was designed to model a range of typical object-oriented database applications. The database consists of several *modules*, which are tree-based data structures. The leaves of the trees each contain a randomly connected graph of 20 objects. In our experiments we used the "small"-sized database according to the OO7 benchmark specification. Each OO7 transaction performs 10 random traversals on either the *shared* module or a *private* module specific to each client. When the traversal reaches a leaf of the tree, it performs either a read or a write action on each of the 20 objects. These transactions are relatively heavyweight when compared to many current benchmarks; each transaction reads about 460 persistent objects and modifies up to 200 of them. If implemented in a straightforward way with a key–value store, each transaction would perform hundreds of get and put operations. Transactions in the commonly used TPC-C benchmark are roughly an order of magnitude smaller [92], and in the YCSB benchmarks [96], smaller still.

OO7 stresses a database's ability to handle read and write contention, because its transactions are relatively large, and because of the tree structure of the database. However, since updates only occur at the leaves of the tree, writes are uniformly distributed in the OO7 specification. To better model updates to popular objects, we modified traversals to make read operations at the leaves of the tree exhibit a power-law distribution with $\alpha = 0.7$ [16]. Writes to private objects were also made to be power-law distributed, but remained uniformly distributed for public objects.

9.4. Experimental setup

We ran our experiments on two different system configurations: one for demonstrating the performance of Fabric using OO7 and CMS, and another for measuring the performance effects of mobile code using FriendMap and BiddingAgent.

When measuring performance of distributed servers, the choice of load model has been shown to have a large effect [86]. Often performance has been measured with a *closed* system model, in which the number of clients generating requests is fixed. In an *open* system model, user requests arrive independently at some average rate. This is usually considered more realistic and a better way to evaluate system scalability because it models the burstiness of requests in real systems. Our OO7 and CMS performance experiments use a compromise that we call a semi-open system model. In the open model, worker nodes execute transactions at exponentially distributed intervals while meeting a specified *average request rate*. Consequently, each worker is usually running many transactions in parallel. Overall system throughput is the total throughput from all workers. Our *semi-open* model follows the open model, but due to experimental resource constraints, there are a fixed number of workers, each with limited capacity to run concurrent transactions. Under extreme system loads, a worker might be unable to start a new transaction (due to too many concurrent threads), and the load generated would therefore be lower than that of a truly open system. To find the maximum system throughput, the average request rate is increased slowly until the target throughput cannot be achieved.

System	Stores	Tput (tx/s)	Latency (ms)
JPA	1	72 ± 12	211 ± 44
Fabric	1	3032 ± 144	143 ± 120
Fabric	3	4090 ± 454	311 ± 175

Fig. 21. CMS throughput and latency on various systems. Reported results are averaged over 10 s at max throughput, across three runs.

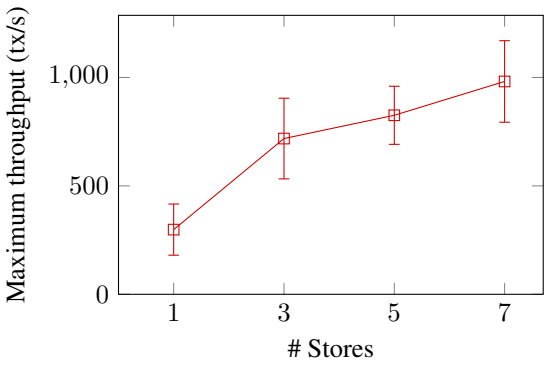


Fig. 22. OO7 maximum throughput on a 2%-write workload as the number of stores increases.

These experiments are run on a Eucalyptus cluster. Each store runs on a virtual machine with a dual-core processor and 8 GiB of memory. Worker machines are virtual machines with 4 cores and 16 GiB of memory. The physical processors are 2.9 GHz Intel Xeon E5-2690 processors.

The performance effects of mobile code were measured in a non-distributed setting, on an Intel Core i7-860 system with 4 GiB of memory.

9.5. Results

Figure 21 reports the performance of the CMS application in various configurations. In the three-store configuration for Fabric, two stores each held data for multiple courses, while the third store contained metadata. We present results only for a single JPA database instance. Even in this single-store setting, and even with Hibernate running in a weaker-consistency mode ("optimistic locking"), Fabric significantly outperforms it in our experiment. Moreover, because the CMS workload is read-biased, we expect JPA with pessimistic locking to perform even worse [89,49].

Although the JPA implementation enforces weaker consistency, Fabric's more precise object invalidation helps performance as contention increases. This comparison shows that the performance of Fabric is competitive with an industry-standard framework offering a transactional language-level abstraction.

Increases in throughput would be less compelling if they came at the cost of high latency. Figure 21 also reports the latency measured with the CMS workload on the various systems, showing that they are comparable.

We evaluated the scalability of Fabric using the OO7 benchmark with varying numbers of stores. Recall that the OO7 database consists of several tree-like modules: a private module for each client, and a shared module. In our setup, a "shared store" was reserved for the modules' inner nodes. The leaves of the shared module were also placed on the shared store, while the leaves of each private module were distributed evenly across the remaining stores. Results presented are the average of three runs.

Figure 22 shows maximum throughput in total transactions committed per second by 36 workers, as the number of stores increases. Error bars show the standard deviation of the measurements. The large variation appears to be result of garbage collection. Since the shared store is a bottleneck, this benchmark is decidedly not a best case for measuring scalability. However, Fabric is able to add roughly 130 tx/s per

	Execution time (ms)	
	FriendMap	BiddingAgent
Java class loader	1	2
Bytecode cache	42	217
Deserializing	6	3
Compiling	15,514	8,693
Loading	3	6
Total	15,566	8,921
Downloaded code size	58 kB	19 kB

Fig. 23. Execution time for the steps required to verify and load dynamically compiled mobile code, averaged over five runs.

	Total load time for all classes (ms)	
	FriendMap	BiddingAgent
Dynamically compiled	15,600	9,188
Locally cached	26	298
Non-mobile	—	20

Fig. 24. Total time spent in the class loader, under different conditions. We present the mean over five runs.

	Total execution time (ms)	
	FriendMap	BiddingAgent
Uncached	17,995	14,210
Uncached w/o compile	2,395	5,022
Cached	4,910	5,999
Non-mobile	—	4,873

Fig. 25. Running time of the mobile-code examples

additional store.[20] The plot only counts committed transactions; the percentage of aborted transactions at maximum throughput ranges from 2% to 6% as the number of stores increases from 3 to 7.

To show the effect of mobile code on performance, we break down the execution time of the FriendMap and BiddingAgent examples. Figure 23 gives the execution time of each step required to load the mobile classes from Fabric into the JVM.

As expected, almost all the time is spent invoking the compiler to analyze the code and generate bytecode. Our compiler has not yet been optimized for run-time compilation, so we expect that that time could be reduced significantly. Additionally, a lower-level intermediate representation such as annotated bytecode, would likely support verification with less overhead. Developing such a representation and analysis was beyond the scope of our research goals. More importantly, the analysis can often be avoided entirely—when either the worker has compiled the class in the past and retains a cached copy, or the worker trusts the provider of the code to correctly compile it. The implementation currently only leverages the former case.

To demonstrate this, Figure 24 shows the time required to load the classes in our examples, in two scenarios: with all classes dynamically compiled at load time, and with all classes pre-compiled and locally cached. We have also written a non-mobile version of the bidding agent example, and give the load time for that as well. These results show that the initial cost of dynamically compiling code is quickly amortized when code is used more than once. Sharing pre-compiled code among trusted nodes would reduce overhead even further.

[20]While Figure 22 shows the system scales roughly linearly with the number of stores, we do not believe this continues indefinitely. Unfortunately, generating sufficient throughput to explore scalability beyond seven stores exceeded our available computing resources.

To focus on the run-time overhead of mobile code, we ran the benchmarks both with and without caching compiled bytecode locally. The results of this benchmark are shown in Figure 25.

For comparison, the second row of Figure 25 gives the execution time for uncached code, excluding the dynamic compilation time. For both FriendMap and BiddingAgent, this result is less than the cached execution time (third row of Figure 25). This shows that compiling code has side effects, such as populating the object cache, that would have been performed anyway. The benefits of these compilation effects may vary from application to application, however. FriendMap appears to benefit from compilation effects significantly more than BiddingAgent does.

10. Related work

Fabric provides a higher-level abstraction for programming distributed systems. Because it aims to help with many different issues, including persistence, consistency, security, and distributed computation, it overlaps with many systems that address a subset of these issues. However, none of these prior systems addresses all the issues tackled by Fabric.

Fabric fits into a substantial history of efforts to integrate information flow control into practical language-level programming abstractions; prior systems include SPARK/Ada [9], Jif and Jif/split [64,67,98,103], FlowCaml [88], Laminar [82], Aura [45], Swift [23], LIO [**?**], Jeeves [**?**], and Paragon [**?**]. These previous systems are either not distributed, or provide much more limited control over distributed computation. Many of the contributions of Fabric arise from fully extending information flow methods into the realm of distributed computation over persistent data, where we have encountered new side channels and uncovered new connections between notions of integrity and authority.

OceanStore [79] shares the goal with Fabric of a federated, distributed object store, but focuses more on storage than on computation. It provides consistency only at the granularity of single objects, and does not help with consistent distributed computation. OceanStore focuses on achieving durability via replication. Fabric stores could be replicated but currently are not. Unlike OceanStore, Fabric provides a principled model for declaring and enforcing strong security properties in the presence of distrusted workers and stores.

Jif/split [98], SIF [24], and Swift [23] are prior distributed systems with mutually distrusting nodes, but with more limited goals than Fabric. While these prior systems use language-based security to enforce strong confidentiality and integrity, they do not allow new nodes to join the system, and they do not support consistent, distributed computations over shared persistent data.

DStar [100] controls information flow in a distributed system using run-time taint tracking at the OS level, with Flume-style decentralized labels [52]. Like Fabric, DStar is a decentralized system that allows new nodes to join, but unlike Fabric, it does not require certificate authorities. DStar has the advantage that it does not require language support, but it also controls information flow more coarsely. DStar does not support consistent distributed computations, data shipping, or mobile code. It also has no notion of code integrity or secrecy.

Some previous distributed storage systems have used transactions to implement strong consistency guarantees, including Mneme [62], Thor [57] and Sinfonia [2]. Cache management in Fabric is inspired by that in Thor [19]. Fabric is also related to other systems that provide transparent access to persistent objects, such as ObjectStore [54] and GemStone [17]. These prior systems do not focus on security enforcement in the presence of distrusted nodes, and do not support consistent computations spanning multiple compute nodes.

Distributed computation systems with support for consistency, such as Argus [56] and Avalon [40], usually have not offered single-system view of persistent data, and none enforce information security. Emerald [12] gives a single-system view of a universe of objects while exposing location and mobility, but does not support transactions, data shipping or secure federation. InterWeave [91] is a persistent distributed shared memory system that synthesizes data- and function-shipping similarly to Fabric, and allows multiple remote calls to be bound within a transaction, remaining atomic and isolated with respect to other transactions. However, it does not appear feasible to build a system like Fabric on top of InterWeave, because InterWeave has no support for information security and its mechanisms for persistence and concurrency control operate at the granularity of pages. The work of Shrira et al. [87] on exo-leases supports nested optimistic transactions in a client–server system with disconnected, multi-client transactions, but does not consider information security. MapJAX [68] provides an abstraction for sharing data structures between the client and server in web applications, but does not consider security. J-Orchestra [**?**] creates distributed Java programs by partitioning programs among assigned network locations. Standard Java synchronization operations are emulated across multiple hosts, but neither security nor persistence is considered. Other recent language-based abstractions for distributed computing such as X10 [85] and Live Objects [75] also raise the abstraction level of distributed computing but do not support persistence or information-flow security.

Some distributed storage systems such as PAST [81], Shark [3], CFS [26], and Boxwood [59] use distributed data structures to provide scalable file systems, but offer weak consistency and security guarantees for distributed computation.

IFDB [**?**] provides a SQL-based interface to a single persistent database while tracking information flow fully dynamically. It is not a federated system like Fabric, nor does it provide type-level integration in the language.

Many previous languages [47**? ?** ,61] have explored integrating abstractions for authorization and access control into the programming model. However, these languages do not integrate reasoning about information flow and rely on the programmer to use these abstractions appropriately to enforce security.

UrFlow [22] enforces information flow control in web applications with policies expressed by SQL queries. UrFlow prevents implicit flows in application code, but not those introduced by the queries themselves.

Hails [**?**] dynamically enforces information flow control for Haskell web applications. Like Fabric applications, Hails web apps compose mutually untrustworthy components that may access persistent data. However, Hails components implement a model–view–controller design pattern and may not invoke each other directly, though multiple view–controllers may share the same model. Hails does not prevent read channels, but does prevent termination and timing channels [**?**].

Cross-origin resource sharing (CORS) [**?**] extends the same-origin policy to allow web sites to specify domains that may load resources from other origins. A browser implementing the CORS API performs a "preflight request" to determine what restrictions apply to a resource before fetching the resource. The CORS API does not protect against read channels: preflight requests may leak information from the requesting page.

Fabric's support for secure mobile code can be compared to proof-carrying code (PCC) [69], a general mechanism for transmitting proofs about code to code consumers. Fabric does not contain a general proof checker; clients check code they receive using the Fabric type system. The Fabric approach is analogous to the bytecode verifier used by Java [55], which similarly type-checks JVM bytecode.

Various attempts have been made to strengthen isolation guarantees for JavaScript. Chugh et al. [25] dynamically check loaded code against statically identified residual information-flow requirements. Con-

script [60] applies aspects to JavaScript primitives, isolating loaded scripts in useful ways. Caja [61] provides isolation in web mashups by using capabilities to protect access to resources at a fine granularity. Secure information flow can be enforced by checking capabilities at statically predetermined locations [10], assuming a static analysis of information flow. Hedin and Sabelfeld [**?**] dynamically enforce secure information flow within a JavaScript DOM tree. Securing mobile code in Fabric has similar challenges to securing JavaScript, but Fabric's mobile code may express more general computations, including creating and accessing persistent data, and may communicate with arbitrary nodes.

System extensibility and evolution has been explored in many contexts. To our knowledge, Fabric's mobile code support is the first to address the information security of the assembly and evolution of components in a general distributed setting.

SPIN [39] is an extensible operating system that allows core kernel functionality to be dynamically specialized by modules written in Modula-3. Like Fabric, SPIN leverages language-level features—such as interfaces and type safety—to provide isolation for untrusted system modules. Unlike Fabric, SPIN uses namespace isolation to control access to system resources: capabilities are implemented as references to system resources, with a type capturing access privileges. In contrast, name resolution in Fabric is orthogonal to security, and the security implications of linking with low integrity code are captured by the type system.

Prior work on expressive module systems explored several approaches to component reuse and evolution. Unit [33] and Knit [78] are component definition and linking languages that enable programmatic assembly of components. Composite units are assembled out of smaller ones, and some architectural properties are checked, such as type consistency (in [33]) or user-defined constraints (in [78]). These systems provide more flexible control of namespaces, but they do not address the security of the produced code.

Codebases have similarities to the classpath entries in JAR files [73]. These references are neither versioned nor immutable, so the meaning of Java classes can change over time. JAR files allow packages to be *sealed*, to control who can insert classes into them. Sealing is orthogonal to our consistency requirements: it does not ensure that classes are named consistently nor that the meaning of code is fixed.

11. Lessons learned and future work

The Fabric project has aimed to develop a practical and effective language and platform for building secure distributed systems. A large part of our motivation was a desire to learn whether the principles of information flow could be put into practice for building secure distributed systems, and to gain insight about where future work should focus. The project was valuable on both fronts.

We chose to base the Fabric programming language closely on the existing Jif programming language and on its underlying security model, the DLM. By developing Fabric, we deepened our insights into how to design security models and to connect them to a programming language. One key insight was the identification of a trust ordering on labels, along with a trust relationship between principals and labels. This entirely novel ordering simplified our thinking about security enforcement in distributed systems. It also led to the realization that the label model can be simplified by unifying the concepts of principals and labels, as was done in our follow-up work on the flow-limited authorization model (FLAM) [**? ?**]. This unification also dispenses with an unnecessary distinction between authority and integrity. FLAM would be a simpler and more expressive basis for implementing a system like Fabric.

One challenge of Fabric programming is that Fabric allows the creation of references to objects at less trusted locations—a capability shared by many other distributed systems such as the Web. However,

programs are not very robust unless programmers take care to avoid dangling pointers and related security vulnerabilities. In subsequent work [**?**], we formalized these vulnerabilities and designed a type system to control them.

The decentralized security principle has been invaluable in guiding the design of Fabric. For instance, computing systems—especially distributed systems—tend to be full of side channels. But not all information channels are security vulnerabilities. The decentralized security principle was a crucial guide for distinguishing harmless and harmful side channels. We have largely ignored timing channels in our work on Fabric, but their presence in Fabric motivated subsequent research [7,101,102]. Dynamic authorization checks in Fabric are another form of side channel, which motivated our work to control these side channels in FLAM [**? ?**]. Careful analysis of the transaction protocols in Fabric showed us that transaction aborts could create an exploitable side channel; recent work has shown how to constrain Fabric transactions to eliminate this side channel [**?**].

The decentralized security principle also helped us identify situations in which security mechanisms were needed—or not needed, because security already rested on sufficient trust. Of course, such reasoning is only possible because Fabric makes trust policies explicit. Although the decentralized security principle was constantly useful, we still lack a satisfactory formalization for it. This is surely an interesting area for future work.

The programming model of Fabric supports concurrent but single-threaded transactions. This is a good fit for OLTP-style applications such as web applications. By supporting only single-threaded transactions, we avoid the challenges of controlling internal timing channels [84]. However, many modern distributed systems rely on highly parallel computation distributed across many host machines, as in the MapReduce pattern [**?**]. An interesting direction for future research would be to support such computations while still enforcing strong information security.

In many modern distributed applications, availability and latency are improved by replicating state at widely separated sites. However, performance is improved by weakening consistency guarantees. Fabric provides a strong consistency guarantee, strict serializability. Subsequent work on *warranties* [**?**] shows that performance can be improved by enforcing a semantic notion of strict serializability rather than the usual notion of serializability at the level of read and write operations [77]. Fabric also allows applications to replicate data manually by creating multiple copies of objects and synchronizing them. It may be useful to expose weaker consistency models explicitly in the programming model, so that application writers do not need to invent their own consistency protocols. Since weak consistency is used partly to improve availability, it is interesting to contemplate enforcement of not just confidentiality and integrity but also availability. Efforts in this direction [**?**] have only begun to address the problem.

One obstacle to adoption of Fabric is that its backing store is not compatible with standard SQL-based database administration tools. In principle, a SQL database could be used as a backing store; this change would turn Fabric into a security-enhanced object–relational mapping system. IFDB [**?**] may be a good starting point for implementing such a system without losing security guarantees at the database layer.

The Fabric intermediate language, FabIL, is very useful for building trusted code but has some limitations that are inherited by Fabric. A useful research direction would be to design a better intermediate language for distributed programming, perhaps one that supports features such as concurrency, weak consistency, and replication.

12. Conclusions

We have explored the design and implementation of Fabric, a new, general platform for secure sharing of information and computation resources. Fabric provides a high-level abstraction for secure, consistent, distributed general-purpose computations on persistent, distributed information. Persistent information is conveniently presented as language-level objects connected by pointers. Mobile code can be dynamically downloaded and used securely by applications, subject to policies for confidentiality and integrity. Fabric exposes security assumptions and policies explicitly and declaratively. It flexibly supports a range of computation styles moving code to data or data to code. Results from implementing complex, realistic systems in Fabric, such as CMS and SIF, suggest it has the expressive power and performance to be useful in practice.

Fabric's security model is based on information flow control, which makes it inherently compositional, even in a decentralized system. Fabric's provider-bounded label checking preserves this compositional security assurance even in the presence of mobile code. As a result, code and data from different, partially trusted sources can be combined while providing relatively strong security assurance.

Fabric embodies several important technical contributions. Fabric extends the Jif programming language with new features for distributed programming, while showing how to integrate those features with secure information flow. This integration requires a new trust ordering on information flow labels, and new implementation mechanisms such as writer maps, distributed transaction logging, and hierarchical two-phase commit. The mobile-code architecture is an interesting and useful component in its own right; provider-bounded verification should be a useful technique for securing other mobile-code systems.

Fabric goes farther toward the goal of securely and transparently sharing distributed resources than prior systems. While it is already a usable platform that increases security assurance over the previous state of the art, there are still hard problems left to solve. Its security guarantees, particularly those regarding confidentiality, are weakened by the presence of side channels—though these side channels are also present in other systems. And while Fabric is larger than other software systems whose functionality has been formally verified, formal verification represents an important future goal. Despite these limitations, Fabric succeeds in offering both a simple, general abstraction for building secure systems and an implementation that can be used to build real applications with stronger security assurance than in any previous platform for distributed computing.

Acknowledgments

We thank Ethan Cecchetti, Chinawat Isradisaikul, Tom Magrino, Yizhou Zhang, and the reviewers for many helpful suggestions that improved the presentation of this paper. This work was supported by MURI grant FA9550-12-1-0400, by NSF grant 1513797, and by gifts from Infosys and Google.

References

[1] jQuery JavaScript framework. http://jquery.com/.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, pages 159–174, Oct. 2007.

[3] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, May 2005.

[4] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE Symp. on Security and Privacy*, pages 191–205, May 2012.

[5] A. Askarov and A. C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), Sept. 2011.
[6] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, Oct. 2008.
[7] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. 17th ACM Conf. on Computer and Communications Security (CCS)*, pages 297–307, Oct. 2010.
[8] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. International Conference on Deductive Object Oriented Databases*, Kyoto, Japan, Dec. 1989.
[9] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, Apr. 2003. ISBN 0321136160.
[10] A. Birgisson and A. Sabelfeld. Capabilities for information flow. In *Proc. 6th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, June 2011.
[11] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 217–230, Dec. 1993.
[12] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proc. 1st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 78–86, Nov. 1986.
[13] H. Böck. *Java Persistence API*. Springer, 2011.
[14] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, 2004.
[15] C. Botev, H. Chao, T. Chao, Y. Cheng, R. Doyle, S. Grankin, J. Guarino, S. Guha, P.-C. Lee, D. Perry, C. Re, I. Rifkin, T. Yuan, D. Abdullah, K. Carpenter, D. Gries, D. Kozen, A. Myers, D. Schwartz, and J. Shanmugasundaram. Supporting workflow in a course management system. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 262–266, 2005.
[16] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.
[17] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Comm. of the ACM*, 34(10): 64–77, Oct. 1991.
[18] M. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proc. 9th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 414–426, 1994.
[19] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, Oct. 1997.
[20] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1 (2), 1983.
[21] W. W.-Y. Cheng. *Information Flow for Secure Distributed Applications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug. 2009.
[22] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.
[23] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, Oct. 2007.
[24] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security Symp.*, Aug. 2007.
[25] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2009.
[26] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symp. on Operating Systems Principles (SOSP)*, Oct. 2001.
[27] M. Day, B. Liskov, U. Maheshwari, and A. C. Myers. References to Remote Mobile Objects in Thor. *ACM Letters on Programming Languages and Systems*, Mar. 1994.
[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st SOSP*, 2007.
[29] L. G. DeMichiel. *Enterprise JavaBeans Specifications, Version 2.1*. Sun Microsystems.
[30] J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3): 143–155, Mar. 1966.
[31] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.1. Internet RFC-4346, Apr. 2006.
[32] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *In Proc. IEEE Workshop on Hot Topics in Operating Systems*, Schoss Elmau, Germany, May 2001.
[33] M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, May 1998.

[34] R. J. Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, Dept. of Computer Science, University of Washington, Seattle, WA, Dec. 1985.
[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.
[36] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.
[37] J. N. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.
[38] T. M. Greanier. Flatten your objects: Discover the secrets of the Java Serialization API. *JavaWorld*, July 2000.
[39] R. Grimm and B. N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Transactions on Computer Systems*, 19(1):36–70, Feb. 2001.
[40] M. Herlihy and J. Wing. Avalon: Language support for reliable distributed systems. In *Proc. 17th International Symposium on Fault-Tolerant Computing*, pages 89–94. IEEE, July 1987.
[41] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. Technical Report CMU-CS-88-120, Carnegie Mellon University, Pittsburgh, Pa., 1988.
[42] R. Housley, T. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Internet RFC-3280, Apr. 2002.
[43] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW '07*, May 2007.
[44] JavaSoft. Java Remote Method Invocation . `http://java.sun.com/products/jdk/rmi`, 1999.
[45] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *Proc. 13th ACM SIGPLAN Int'l Conf. on Functional Programming*, Sept. 2008.
[46] jmeter. JMeter. `http://jmeter.apache.org`.
[47] A. K. Jones and B. Liskov. A language extension for expressing constraints on data access. *Comm. of the ACM*, 21(5): 358–367, May 1978.
[48] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM TOCS*, 6(1):109–133, Feb. 1988.
[49] G. King et al. Hibernate developer guide. Hibernate Community Documentation. `http://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html`.
[50] L. T. Kohn, J. M. Corrigan, and M. S. Donaldson, editors. *To Err is Human: Building a Safer Health System*. The National Academies Press, Washington, D.C., Apr. 2000.
[51] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *CCS '07*, Oct. 2007.
[52] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating System Principles*, 2007.
[53] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.
[54] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of the ACM*, 34(10):50–63, Oct. 1991.
[55] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*, 1999.
[56] B. Liskov. The Argus language and system. In *Distributed Systems: Methods and Tools for Specification; An Advanced Course*, volume 190 of *Lecture Notes in Computer Science*, pages 343–430. Springer-Verlag, Berlin, 1985.
[57] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 318–329, June 1996.
[58] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, 2009.
[59] J. MacCormick, N. Murph, M. Najor, C. A. Thekkat, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.
[60] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, May 2010.
[61] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008.
[62] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Office Information Systems*, 8(2): 103–139, Mar. 1990.
[63] A. C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, Jan. 1999. Ph.D. thesis.
[64] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[65] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997.
[66] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000.
[67] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, , July 2006.
[68] D. Myers, J. Carlisle, J. Cowling, and B. Liskov. Mapjax: Data structure abstractions for asynchronous web applications. In *Proc. 2007 USENIX Annual Technical Conference*, June 2007.
[69] G. C. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Jan. 1997.
[70] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proc. 12th International Conference on Compiler Construction*, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag. LNCS 2622.
[71] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. USENIX Annual Technical Conference*, 1999.
[72] *The Common Object Request Broker: Architecture and Specification*. OMG, Dec. 1991. OMG TC Document Number 91.12.1, Revision 1.1.
[73] Oracle Corp. JAR file specification, 1999. http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html.
[74] Oracle Corporation. Java SE 7 remote method invocation (RMI)-related APIs & developer guides. http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/, 2011.
[75] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn. Programming with live distributed objects. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP)*, 2008.
[76] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006*, Jan. 2006.
[77] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.
[78] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. 4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 347–360, Oct. 2000.
[79] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *2nd USENIX Conference on File and Storage Technologies*, pages 1–14, 2003.
[80] S. Rhea, B. Dodfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of ACM SIGCOMM '05 Symposium*, 2005.
[81] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, Oct. 2001.
[82] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.
[83] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
[84] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.
[85] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Proc. 12th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.
[86] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *Proc. 3rd Conf. on Networked Systems Design & Implementation (NSDI)*, pages 18–31, Berkeley, CA, USA, 2006. USENIX Association.
[87] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proc. ACM/IFIP/Usenix International Middleware Conference (Middleware 2008)*, Dec. 2008.
[88] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.
[89] O. Software. ObjectDB 2.3 developer's guide. `http://www.objectdb.com/java/jpa/persistence/lock`.
[90] T. Stabeil-Kulø and S. Lupetti. Public-key cryptography and availability. In *24th Int'l Conf. on Computer, Safety, Reliability, and Security (SAFECOMP)*, pages 222–232, 2005.
[91] C. Tang, D. Chen, S. Dwarjadas, and M. L. Scott. Integrating remote invocation and distributed shared state. In *Proc. 18th International Parallel and Distributed Processing Symposium*, Apr. 2004.
[92] Transaction Processing Performance Council. TPC-C. `http://www.tpc.org/tpcc/`.
[93] W3C. SOAP version 1.2, June 2003. W3C Recommendation, at `http://www.w3.org/TR/soap12`.
[94] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th SOSP*, pages 203–216, Dec. 1993.
[95] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symp. on Security and Privacy*, pages 52–63, Oakland, California, USA, May 1998.

[96] Yahoo. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[97] M. Zalewski. Browser security handbook, part 2, 2009. http://code.google.com/p/browsersec/wiki/Part2.

[98] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, Aug. 2002.

[99] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[100] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, 2008.

[101] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. 18th ACM Conf. on Computer and Communications Security (CCS)*, pages 563—574, Oct. 2011.

[102] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2012.

[103] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.