Sharing Mobile Code Securely With Information Flow Control

Owen Arden Michael D. George Jed Liu
K. Vikram Aslan Askarov Andrew C. Myers
{owen,mdgeorge,liujed,kvikram,aslan,andru}@cs.cornell.edu
Department of Computer Science
Cornell University

Abstract

Mobile code is now a nearly inescapable component of modern computing, thanks to client-side code that runs within web browsers. The usual tension between security and functionality is particularly acute in a mobile-code setting, and current platforms disappoint on both dimensions.

We introduce a new architecture for secure mobile code, with which developers can use, publish, and share mobile code securely across trust domains. This architecture enables new kinds of distributed applications, and makes it easier to reuse and evolve code from untrusted providers. The architecture gives mobile code considerable expressive power: it can securely access distributed, persistent, shared information from multiple trust domains, unlike web applications bound by the same-origin policy. The core of our approach is analyzing how flows of information within mobile code affect confidentiality and integrity. Because mobile code is untrusted, this analysis requires novel constraints on information flow and authority.

We show that these constraints offer principled enforcement of strong security while avoiding the limitations of current mobile-code security mechanisms. We evaluate our approach by demonstrating a variety of mobile-code applications, showing that new functionality can be offered along with strong security.

1. Introduction

We are entering an era in which code is exchanged rather freely among networked computers. The web has made mobile code a part of everyday life: visiting a web page typically loads JavaScript code from multiple providers onto your system. Web services such as Facebook allow third parties to provide applications that are dynamically combined with their core functionality. Even traditional desktop applications are dynamically downloading plugins from third-party providers.

Over the past few years a whole new 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 copy code and modify it for their purposes; sometimes they write web applications that import code directly via URL. Various popular JavaScript libraries have made this latter use common. Other popular mobile-code platforms, including ActionScript (for Flash) and Java, are not fundamentally different.

While there are many benefits to promiscuously sharing mobile code, these benefits come at a cost: in general, it is hard to determine if dynamically combining code from multiple sources yields a secure application. On the web, the main security safeguard is the *same-origin policy* [35], 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. The limitations on expressive power force developers to work around the same-origin policy [14], potentially introducing additional vulnerabilities. These problems will be exacerbated as applications begin relying on persistent storage functionality provided by the recent HTML 5 specification [12].

The goal of this work is to create a platform that securely supports the flexible use and reuse of mobile code that developers clearly want. Our contribution is a system that provides expressive power while protecting the confidentiality and integrity of information—even though sharing happens between sites lacking mutual trust. We take a holistic view of security: not only must each individual application component be secure, but also the entire assembly of code and data from various providers must satisfy all participants' security requirements.

By their very nature, isolation mechanisms such as the same-origin policy prevent sharing. In response to the need for sharing, programmers inevitably open communication channels across isolation boundaries, reopening similar security vulnerabilities in new guises. The usual response is to attempt to control these channels using authorization mechanisms such as capabilities. This attempt is doomed to fail. The problem is that authorization is not compositional, because it does not take into account *what* information is being communicated—authorization mechanisms allow *any* communication as long it is performed by an authorized principal. Compositionality is needed for federated environments where code and data from different sources are combined.

Information-flow control is an appealing alternative because it is inherently compositional. Information-flow control mechanisms and policies have been developed for reasoning about confidentiality and integrity in decentralized systems [24, 25, 36, 40, 37, 38, 30, 20]. These systems ensure that the flow of information is in accordance with policies expressed as labels on resources. Enforcement is done through some combination of static or dynamic mechanisms at either the programming language level or at the operating system level, and the use of these mechanisms can be proved to

ensure strong security properties [31]. However, these prior systems do not support secure sharing of mobile code.

In our new mobile-code architecture, code becomes a persistent resource managed by the system. The system allows principals to publish code and tracks how much trust has been placed in code by principals in the system, automatically bounding the actions and privileges of the code according to that trust. In addition, to support reuse and adaptation of existing code, as happens in the web today, our new mobile-code architecture provides mechanisms for secure evolution of code and persistent data.

We have built a prototype of our secure mobile-code platform as an extended version of the prototype Fabric system [20, 19]. Fabric is a decentralized computing platform that provides a high-level, object-oriented programming model for secure distributed computation. While Fabric already addressed many distributed security issues, it did not support mobile code. To distinguish the new version of Fabric from the original, we refer to it as Mobile Fabric.

The advantage of working in the context of Fabric, rather than, say, in JavaScript, is that we can solve a more general, more abstract instantiation of the mobile-code problem. Working within the context of existing web standards would obscure rather than illuminate the fundamental issues of sharing untrusted and partially trusted code. Moreover, the lessons learned from this work should be applicable to web security as well; for example, the constraint placed on code in Mobile Fabric might inform emerging cross-origin policies [33], where attempts are being made to obtain some of the expressive sharing obtained by Mobile Fabric.

The rest of the paper is structured as follows. Section 2 introduces a running example to demonstrate both the new functionality that secure mobile code offers and the new challenges it presents. Section 3 provides background on decentralized information-flow control (DIFC) and its instantiation in the Fabric system. Section 4 describes the overall architecture, including the threat model, and Section 5 explains the new mechanisms required to securely execute mobile code. Section 6 discusses mechanisms to support secure evolution of mobile code. Section 7 describes the prototype implementation of the new architecture, and Section 8 presents performance and experience results. Section 9 discusses related work, and Section 10 concludes.

2. A mobile-code example: untrusted mashups

The "FriendMap" application illustrates the security challenges of allowing untrusted mobile code to operate on sensitive data. This application enables a user of a social network to create a map displaying the locations of their friends. Let us call the user "Alice" and one of her friends "Bob." Figure 1 shows the interactions Alice's client makes while executing FriendMap. First, Alice's client downloads the application code (1) and executes it locally. FriendMap then fetches the locations of Alice's friends (2) from the social network ("Snapp"), requests a map (3) from a third-party map service ("MapServ"), and plots the friends' locations on the map displayed to the Alice. Alice can also choose to post the map (4) to the social network to share with her friends.

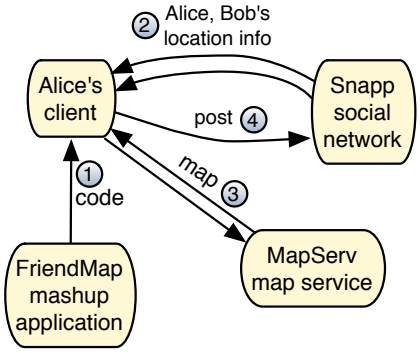


Fig. 1: Overview of the social mashup example

2.1. Security considerations

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

Further, none of the involved principals trust the provider of the FriendMap code. Therefore some mechanism is needed to ensure that the code enforces their 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 with the confidentiality of his location, so Alice's node is responsible for enforcing this confidentiality policy.

In real applications, policies are more nuanced than lists of entities allowed to learn information. Bob may consider his exact location confidential, but not his current zip code. This is an example of declassification: some information about Bob's location is released, even though Bob's precise location is secret. The decision to declassify must be authorized by Bob, so the code performing the declassification must either be provided by or endorsed by Bob.

Existing platforms for secure mobile code fail to meet the security requirements of FriendMap. Isolation-based approaches, such as the same-origin policy [35] and SFI [34], entirely prevent applications from interacting with each other. FriendMap would be unable to access the locations of Alice's friends, regardless of the policies on those locations. Authorization approaches such as OAuth [11] suffer from the opposite problem: once Alice's client is able to see the locations, nothing prevents FriendMap from leaking the data

to MapServ, her personal bulletin board, or even FriendMap's developers. By contrast, Mobile Fabric provides abstractions and mechanisms to meet FriendMap's security requirements.

2.2. Software construction and evolution

Reusing code improves productivity, interoperability, and performance. But reusing and composing partially trusted code and data presents new challenges. Developers need assurance that their assumptions regarding a program's dependencies are met, particularly when the program manipulates sensitive data. For instance, a developer might only trust particular library vendors or particular library versions to handle sensitive data. If a different library is used at run time, the resulting program may be insecure or incorrect.

The developers involved in a collection of interdependent code may have made conflicting assumptions about what names in their code mean. Discovering and fixing such errors at run time can be very difficult. Therefore, Mobile Fabric includes support for detecting conflicting assumptions about the binding of names in code (in particular, class names) to actual code objects stored persistently in the system.

The problem of dependencies is made more complex because real software evolves over time and because in Fabric, code is tied to persistent objects whose behavior it defines. As software evolves, changes to code may require replacing or updating objects. In many updates, only a small percentage of the code actually changes. Therefore, the goal of Mobile Fabric is to allow the reuse of compiled code and persistent objects without introducing inconsistencies.

Code sometimes evolves in a way that requires updates to persistent data. In these cases it is important to be able to migrate the data from one version to another, even though the code for the old version may resolve names in a way that is inconsistent with the new version.

As we show in Section 6, Mobile Fabric provides the flexibility to securely and incrementally evolve and extend programs constructed from software components from multiple trust domains. It supports evolution of code and data, while avoiding unnecessary updates to both.

3. Background: DIFC and Fabric

Mobile Fabric extends the Fabric system [20] with support for mobile code. Fabric provides a high-level, language-based abstraction for constructing secure distributed applications. Its principled security enforcement makes Fabric a good starting point for mobile code. However, Fabric's security mechanisms, both at the language and the system level, do not handle code that might be supplied by an adversary.

Some additional background on Fabric will be helpful. Fabric's language model for distributed programs treats every resource—distributed or local, persistent or non-persistent—as an object in a Java-like object-oriented language. Compile-time and run-time mechanisms protect the confidentiality and integrity of information in Fabric by controlling information flows. Flows within locally executed code are checked largely at compile time. Run-time checking is primarily used at remote calls to verify that the remote host node can be trusted to enforce the information-flow policies of the call.

3.1. Objects

Principals, policies, remote hosts, local data, and remote data are all language-level objects in Fabric, and objects may reference any other object. Each object is given a unique object identifier, or oid. An oid includes the host storing the object, so an object can be located using only its oid, and oids can be exported outside of Fabric.

3.2. Nodes

Fabric nodes serve different roles. Objects are stored persistently on *storage nodes* (or just *stores*); computation takes place on *worker nodes* (or *workers*). Workers fetch objects from stores. The web analogue of workers and stores would be clients and servers.

Fabric can express different styles of distributed computation, because a single computation can span both multiple workers and multiple stores. This model is more general than web applications, where computations cannot span multiple client browsers. By default, Fabric computations use *data shipping*: when computation at a worker needs a remote object, the object's store sends a copy of the object to the worker. The worker pushes updates back to the object's store when the computation completes. In the FriendMap example, Snapp and MapServ store objects used by the FriendMap application. Alice's worker initiates and coordinates the computation, which includes 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.

3.3. Delegating Trust

Fabric represents entities involved in a computation as abstract *principals*. Principals may represent a single entity such as a user or a Fabric node, or multiple entities such as an organization or a group of friends.

A principal p may express trust in another principal q by allowing q to *act for* it. In this case we write $q \succcurlyeq p$, which we read as "q acts for p". We also sometimes say that "p delegates to q", or "p trusts q". It is essentially the same as $q \Rightarrow p$ in many authorization logics (for example, [17]).

If $q \succcurlyeq p$, then q may perform any action that p may perform. For example, q may read any data that p can read, or update any data that p may update. Principal q may also downgrade policies owned by p.

In the FriendMap example, `Snapp`, `MapServ`, and `FriendMap` are all principals, as are users, such as `Alice` and `Bob`. The ability to create new principals allows for flexible, fine-grained delegation. As shown in Figure 2, Snapp users do not trust their friends' principals. Instead, users have a

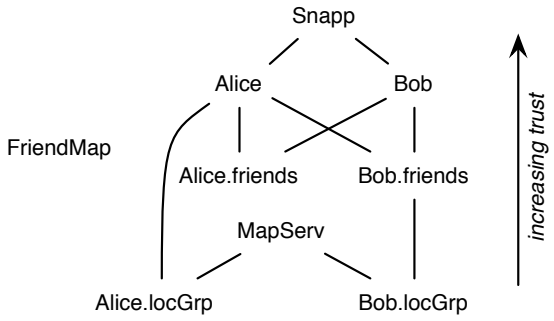


Fig. 2: Trust relationships in the FriendMap example.

separate principal (e.g., `Alice.friends` and `Bob.friends`) representing the group of their friends, and that principal delegates to each friend's principal. Similarly, each user has a principal (e.g., `Alice.locGrp` and `Bob.locGrp`) representing the group of principals who can learn the user's location.

The acts-for relation is transitive: if $p \succcurlyeq q$ and $q \succcurlyeq r$ then $p \succcurlyeq r$. This allows policy changes to be implemented using small changes in the acts-for hierarchy. For example, Bob might decide that all of his friends should be able to read his location. He can implement this by making `Bob.locGrp` delegate to `Bob.friends`, as depicted in Figure 2. Since `Alice` $\succcurlyeq$ `Bob.friends`, this immediately implies that `Alice` $\succcurlyeq$ `Bob.locGrp`, allowing Alice to read Bob's location.

The acts-for relationship is also used to express trust in Fabric nodes. The roots of trust in Fabric are X.509 certificates [13] published by nodes. These certificates include the node's hostname and the oid of its principal object.

The set of principals generates a lattice ordered by $\succcurlyeq$. The *top principal* $\top$ acts for all other principals, and all principals act for the *bottom principal* $\bot$.

3.4. Labels

Fabric programs express security policies using *labels*. Labels are drawn from the decentralized label model (DLM) [25], and express policies in terms of principals. Every object in Fabric has an *object label* describing the confidentiality and integrity of its contents. This label is stored persistently with the object, and constrains how information can flow via the object. Information whose confidentiality is too high or whose integrity is too low cannot be stored in the object, and information from the object cannot flow to a location whose confidentiality is too low or whose integrity is too high. Labels are part of types, and compile-time *label checking* controls information flow within programs with fine granularity.

To support the decentralization of information-flow policies, each policy in a label is owned by a principal. A label is a set of such *owned policies*. For example, Bob might protect information about his location by the confidentiality policy `Bob`$\rightarrow$`Bob.locGrp`. This policy states that Bob owns the information, but he allows the information to flow to the `locGrp` principal (and implicitly to himself).

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

Fig. 3: Labels in the FriendMap example

Integrity policies are also owned: the policy `Alice`$\leftarrow$`Alice.friends` means that Alice owns the data, but that it may be influenced by her friends. This policy is part of the label on Alice's bulletin board on the social network. The labels in the FriendMap example are summarized in Figure 3.

Fine-grained principals like a user's friend group allow other principals to control the meanings of policies by changing the acts-for hierarchy. For example, Bob has allowed all of his friends to read his location by making his `locGrp` principal delegate to his `friends` principal. Alice has not made this delegation, so her friends cannot read her location.

Informally, a principal p is trusted to enforce a label ℓ if it can both see and affect data at that label. We express this with the notation $p \succcurlyeq \ell$. A principal is trusted to enforce a confidentiality policy if it acts for a reader or owner of the policy; it is trusted to enforce an integrity policy if it acts for a writer or an owner of the policy.

3.5. Orderings on labels

The acts-for relation $\succcurlyeq$ gives rise to an *information-flow ordering* ordering on labels. The ordering $\ell_1 \sqsubseteq \ell_2$ captures when flow from label ℓ_1 to label ℓ_2 is secure. We say "ℓ_1 can flow to ℓ_2" or "ℓ_1 is less restrictive than ℓ_2". For example, if `Bob` acts for `Alice.friends`, then `Alice` $\rightarrow$ `Alice.friends` $\sqsubseteq$ `Alice` $\rightarrow$ `Bob` because fewer principals can learn information labeled `Alice` $\rightarrow$ `Bob` than could have learned information labeled `Alice` $\rightarrow$ `Alice.friends`.

The ordering $\ell_1 \sqsubseteq \ell_2$ is used when checking information flow within Fabric programs, both at compile time and at run time. For example, when assigning from a variable with label ℓ_1 to one with label ℓ_2, this ordering must hold.

3.6. Declassification and endorsement

In general, the ordering $\sqsubseteq$ only approximates the true information security requirements of an application, so sometimes it prevents flows that applications need. DIFC systems such as Fabric allow these flows using *downgrading* operations. Declassification is a downgrading operation which removes confidentiality policies; endorsement is one that adds integrity policies.

Fabric programs must explicitly specify declassification and endorsement. Additionally, all policy-downgrading code is marked with an authority clause that specifies the principal authorizing 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 rule enforces *robust downgrading* [2], which prevents the adversary from causing these operations to be misused.

The *program-counter label*, pc [23], identifies the information that influences whether a given program point is reached. The Fabric type system requires that the left-hand sides of assignments have labels higher than pc in the information-flow ordering. For confidentiality, this rule prevents information from leaking through *implicit flows* [8]. It is also crucial for integrity, where it prevents untrusted data from indirectly influencing trusted data.

3.7. Adversaries

Every Fabric principal views every untrusted principal as a potential adversary, so who the adversary is depends on whom you ask. In the FriendMap example, the FriendMap application provider is treated as an adversary since no one trusts 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.

Fabric's goal is to ensure that the security of a principal does not depend on any part of the system that it considers to be an adversary. This is called the *decentralized security principle*. More precisely, the security of an object, expressed by the policies in its label, should only depend on system components that are trusted to enforce those policies. Thus, the integrity of an object with label ℓ should be unaffected by an adversary A unless $A \succcurlyeq I(\ell)$, and its confidentiality should be unaffected unless $A \succcurlyeq C(\ell)$.

In a decentralized system, there is no single *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. An architecture for secure mobile code

The challenge of Mobile Fabric is to maintain the strong yet decentralized security guarantees of Fabric while giving adversaries the power to upload and execute mobile code. To address this challenge, we add several new components to the system architecture, depicted in Figure 4. As in Fabric, information is stored in persistent objects that can refer to each other. Unlike in Fabric, code can also be stored persistently at stores, and downloaded and executed by workers. We will call code stored in Fabric *class objects* (not to be confused with Java class objects). Each object contains a reference to a class object, which defines its structure and behavior.

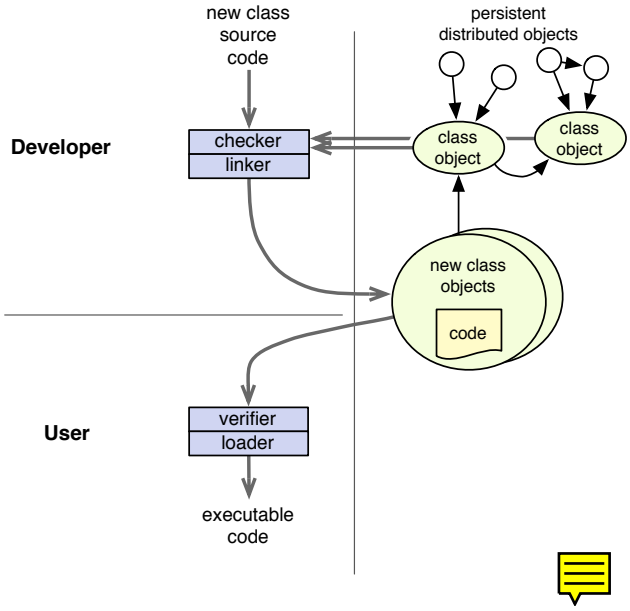


Fig. 4: Compiling, linking and loading mobile code

4.1. Producing mobile code

These new class objects are created and loaded onto stores by Fabric workers. In particular, the Mobile Fabric compiler and linker are themselves Fabric programs that code providers can invoke to turn source code into Fabric objects. This process is depicted in the "Developer" portion of Figure 4.

A novel feature of this design is placing information-flow policies on class objects. Like other objects stored in Fabric, class objects are given labels. These labels are essential for preventing adversaries from using mobile code to compromise integrity and confidentiality.

The code for a class can mention other classes, hence the meaning of a class depends on other classes that it names. Unlike in many other mobile-code platforms such as JavaScript and Java, the binding between names and classes is defined by the code developer when the class is published. We call this linkage specification a *codebase*. Each class object specifies its codebase.

Mobile Fabric extends the Fabric language to enable both compile-time and run-time reasoning about the trust that can be placed in code. Class source can explicitly refer to the provider label of any class, including its own, using the name `provider`. The provider label can appear in label annotations checked at compile time, and can also be compared to other labels at run time. Code is label-checked without making any assumptions about the `provider` label. This forces all assumptions about the code's provider to be made explicit in the source code so that the same source code can be securely reused, relinked, or provided by different principals.

4.2. Executing mobile code

Code is executed at worker nodes. In the original Fabric system, workers stored all code locally and implicitly trusted it. In Mobile Fabric, workers may also fetch code from class objects that are themselves stored in Fabric, as shown in the "User" portion of Figure 4.

Workers load the code for a class when an instance of that class is first fetched from a store or when the class is a dependency of another class being loaded. The worker fetches the class object and verifies the information flows within the class's code. Finally, it converts the class into executable bytecode, thus allowing methods on the class to be invoked.

4.3. Threat model

An important component of any security architecture is the threat model, which defines the power of the adversary. Mobile Fabric allows a powerful adversary—one that is even more powerful than in Fabric. Therefore, security assurance is in some ways stronger than in the prior system.

As discussed in Section 3, any principal may be considered an adversary, so we define the power of adversaries in terms of untrustworthy principals. Since the system should be secure from the viewpoint of every participating principal p, the adversary is some principal A where $A \not\succcurlyeq p$. Therefore, we can analyze security assuming an arbitrary but unspecified principal A. All of the security mechanisms are designed to enforce security regardless of the particular choice of adversary.

From the viewpoint of principal p, a node n might be controlled by the adversary if $n \not\succcurlyeq p$. This untrusted node need not be running the Mobile Fabric implementation and can therefore violate its rules. The node might participate incorrectly in the low-level protocols that implement the system. It might view or leak any keys or messages that adversary-controlled nodes receive, and it can read and update any state on any other node it acts for. We do assume that the adversary cannot fabricate cryptographic signatures or decrypt messages without the appropriate private keys; therefore, trustworthy nodes can communicate with each other over authenticated private channels.

Our goal is to allow secure interaction despite distrust, so adversaries are able to read and update certain objects stored at trustworthy nodes. In particular, a node n allows a node a to read an object with low-confidentiality labels ℓ; that is, if $a \succcurlyeq C(\ell)$. This read is allowed even if n considers a an adversary. Similarly, n allows a to update objects with low-integrity labels ($a \succcurlyeq I(\ell)$).

Class objects containing mobile code are particularly important examples of objects adversaries can affect. Adversaries can provide mobile code to trustworthy nodes, as long as the code has a low integrity label. The challenge is to prevent this code from compromising security.

Adversaries can also try to use remote calls to invoke methods on trustworthy nodes. These calls are allowed only if the initial $\underline{\mathsf{pc}}$ label of the method is low-integrity.

Fabric uses SSL for communication, providing some protection against network adversaries. We therefore 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. Because network destinations are visible in network packets, the adversary might be able to learn 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 might mitigate these channels, though at a cost.

As in most work on distributed system security, timing, termination [31], and progress channels [1] are largely ignored. Termination and progress channels might be justifiably ignored because they have low bandwidth, but timing channels can have high bandwidth. There are two kinds of timing channels: *external* and *internal* [32]. We do not attempt to control covert external timing channels in this work; in other words, adversarial nodes are assumed not to time when messages arrive. Run-time mitigation methods (e.g., [15, 39]), might be useful for limiting the bandwidth of these 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.

5. Securing mobile code

In the original Fabric system, workers trust all code they execute. Since adversaries can provide code to workers in Mobile Fabric, two key assumptions about code are invalidated: that all code goes through type checking and label checking to ensure its information flows are secure, and that code is provided by a trusted entity. Fabric also assumes certain vulnerabilities arising from inconsistent objects and read channels can be ignored safely because they are hard to exploit without the complicity of trusted nodes.

In this section, we analyze the vulnerabilities inherent in systems supporting mobile code and present our solutions for preserving the security of Mobile Fabric using information flow control. These solutions are summarized in Figure 5, which also serves as a roadmap to the remainder of this section.

5.1. Label-checking mobile code

We cannot be sure that code dynamically loaded by workers respects information-flow policies. Therefore, at load time, workers perform static label checking of all dynamically loaded mobile code. Because type checking and label checking are modular, this analysis can be performed whenever a new class is encountered.

Code must be stored by Mobile Fabric in a form that permits accurate static analysis. In the current prototype, code

- Mobile-code label checking (§5.1).
  - adversary-provided code creates no insecure information flows within trustworthy nodes
- Provider-bounded label checking (§5.2)
  - untrusted code cannot corrupt high-integrity information or downgrading decisions
  - confidential code cannot affect public output
- Type fingerprint checking for remote calls (§5.4)
  - information flows caused by remote calls comply with integrity and confidentiality restrictions
- Type fingerprint checking when objects are loaded (§5.5)
  - the dynamic types (including information-flow labels) of objects are consistent with the static type.
- Access label checking (§5.6)
  - information flows caused by fetching objects comply with confidentiality restrictions

Fig. 5: Summary of the novel dynamic checks that trustworthy nodes perform, and the invariants they preserve. These invariants enforce the decentralized security requirements of Mobile Fabric.

```
1 String{user←user} password;
2 void initialize_password{user←user}() {
3   password = "init";
4 }
```

Fig. 6: Mobile code creating a vulnerability

is stored and loaded as source—like JavaScript but unlike Java. Storing code as source is not essential to the design, and other code formats such as abstract syntax trees or annotated bytecode are possible.

Code is trusted if the user completely trusts the code provider. According to the decentralized security principle, the user can load such code without analyzing it. To accelerate loading and execution, trusted code can be loaded as bytecode or even as machine code.

5.2. Provider-bounded label checking

Label checking ensures all flows in code obey the information-flow ordering, but this is not enough to stop adversary-provided code from introducing vulnerabilities. For example, a method like that in Figure 6 type-checks, because only high-integrity information appears to influence the high-integrity variable `password`: the literal `init`, and the fact that the method was called, which is captured by the $\underline{\mathtt{pc}}$ label at the call site. The literal `init` is considered trusted because the code is trusted, whereas $\underline{\mathtt{pc}}$ at the call site is constrained by the initial $\underline{\mathtt{pc}}$ label of the method (the "begin label" [23]), which in this case is trusted: $\{\mathtt{user}\leftarrow\mathtt{user}\}$.

However, if the adversary convinces a trusted worker node to use this code, it might be used to change the password, a clear security vulnerability.

To solve this problem, we extend the program-counter ($\underline{\mathtt{pc}}$) mechanism. With respect to a given adversary A, a *high-confidentiality context* is a part of the program about whose execution the adversary is not trusted to learn: $A \not\succcurlyeq C(\underline{\mathtt{pc}})$. A *low-integrity context* is a part of the program whose execution the adversary can affect: $A \succcurlyeq I(\underline{\mathtt{pc}})$. In either case, security requires restricting how information flows out of the context, so we refer to either sort of context as a *high context*.

The key to preventing attacks like that in Figure 6 is to treat the code itself as information that affects the results it produces. Therefore code is stored with an information-flow label (the *provider label*) that bounds the influence of the adversary on the code. In fact, this label is precisely the object label of the class object.

To constrain untrusted code, we join the provider label into the $\underline{\mathtt{pc}}$ label. This makes sense because the code provider affects the statements that are executed. If the code of Figure 6 is provided by an adversary, its low-integrity provider label will effectively make the $\underline{\mathtt{pc}}$ label low-integrity, preventing any assignments to high-integrity variables such as the password. We refer to this analysis as *provider-bounded label checking*.

The provider label also enforces robust downgrading, because 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 in the system not provided by the adversary.

Provider-bounded label checking would prevent an adversary from changing the password by providing the code, because an adversarial provider can only provide code that operates in a low-integrity context, and the high-integrity assignment to `password` cannot occur in such a context.

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.

The high-confidentiality label on the 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 affected by the code from flowing to untrusted nodes. If providers of confidential code wish to make its results public, the results must be explicitly declassified.

5.3. Provider-bounded authority

Another curb on the misuse of declassification and integrity 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, we ensure that $I(\texttt{provider}) \sqsubseteq \{\top \leftarrow p\}$.

5.4. Fingerprint checking for remote calls

The security of both Fabric and Mobile Fabric relies on the assumption that trustworthy nodes agree on the types of classes. When classes are distributed and stored by trusted system administrators, as all classes in Fabric are, it is reasonable to assume that nodes agree on types. But this assumption is no longer safe when class objects can be provided by untrusted nodes.

To see how this could lead to an attack, consider the scenario where a user operates two trusted nodes, `u1` and `u2`, and `u1` wishes to make a remote call to execute the attacker-provided method `harmless_method` at `u2`.

Assume that the attacker provides two different implementations of `harmless_method` to the two nodes. Suppose that `u1` sees the following:

```
1 void harmless_method() {user→user} {
2 }
```

whereas `u2` receives

```
3 void harmless_method() {user→public} {
4   public_data = true;
5 }
```

Both of these methods type-check. `u1` is willing to make this remote call in a context that would reveal confidential data, because the begin label of `harmless_method` prevents it from having any public side effects. On the other hand, `u2` is willing to execute the method even though it has public side effects, because its begin label requires it to be called in a context that does not reveal any sensitive information.

However, when these two are combined, the type mismatch allows the provider to trick the trusted worker into revealing the user's confidential information. It is not enough that the methods both type-check in isolation; they 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* [5] is sent along with each remote call. The receiver checks that the invoked method has a matching fingerprint. The fingerprint is computed as a secure hash over the entire source code of the method's class, including the source code of any superclasses. This ensures more type agreement than strictly necessary, but the same fingerprint has other uses, so there is no harm in it.

A similar vulnerability can occur if a worker flushes the cache of compiled classes: the adversary could make the type of a method appear to change. To prevent this, the fingerprint of a class must be preserved across cache flushes and checked against the class when it is reloaded.

5.5. Fingerprint checking for object loading

An attacker who can change the class associated with an object can cause nodes to disagree about the labels on the object's fields and methods. 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 ensure that the class accurately describes the object, including security policies on its fields.

5.6. Access labels

When an object is accessed during computation on a worker, but is not yet cached at the worker, the worker must fetch the object data from the node where it is stored. Thus, the contacted node learns that an access to the object has occurred. When the access is a read, we call this side channel a *read channel*.

In the original Fabric system, objects are placed onto stores that can enforce their labels, including their confidentiality. However, this does not prevent read channels. According to this rule, public data can be stored on a low (adversary-controlled) node. But then accesses to the object from a high context would violate confidentiality.

Read channels are not controlled in the original Fabric system, but they become easy to exploit once the adversary can provide mobile code that generates such accesses. Read channels are not a Fabric-specific problem, either—holes in the same-origin policy also permit read channels: for example, via images fetched from ad servers.

We control read channels by extending the programming language. We add to each object a second label called the *access label*. It is a confidentiality-only label that bounds what can be learned from the fact that the object has been accessed. The access label ensures that the object is stored on a node that is trusted to learn about all the accesses to it, and it prevents the object from being accessed from a context that is too high. The access label has no integrity component because there is no integrity dual to read channels.

The access label of an object is declared as part of the label of its fields. Given object label ℓ_u and access label ℓ_a, a label annotation ℓ_u`@`ℓ_a means that the field, and by extension the object, has the corresponding labels.

For example, to declare an object containing public information (in field `data`) that can be accessed without leaking information (according to any principal that trusts node n to enforce its confidentiality), we can write code like this:

```
1 class Public {
2     int {} @ {⊤→n} data;
3 }
```

Even though the information is public and untrusted (label `{}`), objects of this class can be stored only on nodes that are at least as trusted as node `n`. Conversely, if we had 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.

Access labels require two new static checks in Fabric code:

1) The access label on fields allows the compiler to check all reads from and writes to fields to ensure that they occur in a low context. The program-counter label pc must be lower

than the access label (i.e., $\underline{\texttt{pc}} \sqsubseteq \ell_a$) at each field access (read or update). This is in addition to the existing check, inherited from Jif [23], that requires $\underline{\texttt{pc}} \sqsubseteq \ell_u$ at each update.

2) At the point where an object is constructed using `new`, the node at which the object is created must be able to enforce the access label. In Fabric, an object of class `C` is explicitly allocated at a node `n` using the syntax `new C@n(...)`. We require $\underline{\texttt{pc}} \sqsubseteq \ell_a$ and $n \succcurlyeq \ell_a$ at this point in the code, because node n learns about the future accesses to the object.

Access labels also interact with provider-bounded label checking. Recall that the compiler ensures the initial $\underline{\texttt{pc}}$ of methods contain at least as much confidentiality as the label of the code. Therefore, the access label of objects used by confidential code must be at least as high as the confidentiality of the code.

Access labels also introduce a new dynamic check. When a worker fetches an object, the access label bounds how much information is leaked to the object's store. However, if the reference to the object is provided by an adversary, there is no guarantee that the store is trusted to learn that information. Therefore, before the fetch is performed, the worker must check dynamically that the store can enforce the access label.

Mobile Fabric also encounters a new kind of read channel that did not exist in the original Fabric system: class object read channels. Fetching an object may require fetching its class, so the class 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, their replication is harmless in Fabric.

5.7. Other covert channels

Read channels are an example of channels that might be exploited as covert channels [18]. The ability to control mobile code clearly increases the power of the adversary to exploit covert channels.

Covert channels fall into two categories: *storage channels*, in which information is learned by observing the state of the system independent of time, and *timing channels*, in which information is learned from the time at which event occurs. Many systems, especially those that support mobile code, have both kinds of channels.

Fabric controls storage channels that are visible at the language level, such as *implicit flows* [9] that arise from the control flow of programs. Other storage channels exist below the language level of abstraction and are blocked or mitigated by a variety of mechanisms. For example, updates to objects shared across a distributed computation are propagated among worker nodes, but the information channel is mitigated cryptographically.

As described in Section 4.3, we ignore some information channels related to network traffic analysis.

6. Software reuse and evolution

Reusable software components help programmers develop complex applications from smaller, modular fragments of code. Using expressive component architectures comes at a price, however. Many frameworks require complex interface definitions or linkage specifications [29, 26, 10, 28], and conflicting dependencies can result in dynamic linking errors that are difficult to resolve.

The most common mobile code in use today, JavaScript code distributed on the web, is not modular: it provides no isolation between scripts loaded by the same webpage so developers must resolve namespace collisions themselves. Loading components into a single global namespace is overly restrictive [3] and tends to make code unnecessarily bound to specific versions of dependencies [28]. For instance, a JavaScript program that imports one version of a library may be difficult to compose with another program using a different version. Yet version conflicts do not necessarily represent fundamental incompatibilities—the choice of URL may be arbitrary. Software that is compatible with multiple configurations is easier to reuse and compose.

Modularity becomes more complicated in distributed settings where nodes access and update persistent data. Here, a schema defines the structure of persistent data and may evolve over time. Likewise, programs interacting with the data may expect it to be structured according to a specific version of the schema. To ensure persistent data remains accessible, the distributed system (or the programs themselves) must either migrate the data to new schemas or handle it in a backward-compatible way. Modularization helps isolate changes so that more code and data remain compatible with each other.

The root of the problems caused by a lack of modularity is that the *meaning* of code changes in different contexts. Problems such as namespace collisions, dependency conflicts, and data corruption are difficult to avoid without modularity since the assumptions made by each context are often subtle.

We argue that the meaning of mobile code should be fixed at publication. In Mobile Fabric, publishers distribute automatically generated linkage specifications called *codebases* along with published code. Codebases support *decentralized namespaces*; a class's own codebase defines the resolution of its dependencies. This mechanism enables independent nodes to resolve dependencies consistently without resorting to a global namespace.

Importantly, name resolution and namespace isolation in Mobile Fabric are orthogonal to security enforcement, unlike in systems such as Java, JavaScript and SPIN [3]. Information-flow control restricts the use of resources rather than the ability to name them. Linking against high-integrity, high-authority code requires no special privilege; instead, label checking ensures the end-to-end security of linked code.

6.1. Codebases

All class objects published in Mobile Fabric are associated with a *codebase*. A codebase maps from class names to

published class objects, specifying a linkage for the static types used in published components. Since linkage of a component's dependencies is fixed at publication, nodes that download and compile mobile code independently can interact with persistent data and each other robustly.

Loading components dynamically but linking them statically seems crucial for security. It also distinguishes Mobile Fabric from other systems, such as Java, where there is no guarantee that the class linked by the JVM at run time is the same that was compiled against. By associating a codebase with published classes, we make it possible for nodes to agree about types in a decentralized way.

Figure 7 depicts two published classes, `pkg.B` and `pkg.C`, and their respective codebases. To compile and run code stored in the class `pkg.B` with dependency `pkg.C`, the following steps occur. During compilation, the compiler consults the entry for "pkg.C" in `pkg.B`'s codebase. This entry contains a reference to the specific class object used to resolve the dependency. Likewise, when compiling `pkg.C` the compiler uses *its* codebase to resolve dependencies. If `C` was published alongside `B`, the codebase will be the same one as before. On the other hand, if `C` was published separately, then `C`'s codebase will be different and may contain entries not present in `B`'s codebase. Figure 7c shows the latter case. Solid arrows indicate the class object a particular class name resolves to, and dotted lines indicate the home codebase of a class. `pkg.B` uses the codebase `CB2` to resolve its dependencies while `pkg.C` uses `CB1`. `CB1` has a local reference to `pkg.C` while `CB2` contains a remote reference to the same object. Using this process to resolve dependencies, nodes are able to compile compatible versions of `pkg.B` independently.

Rather than forcing developers to create codebases by hand, the Mobile 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. To prevent the adversary from exploiting codebases that violate this constraint, the compiler checks that it holds at link time.

6.2. Namespace consistency

The independent components making up a complete program often share common dependencies. In typical use, these dependencies must resolve identically for all components. For instance, the dependency might define an interface through which components interact. In some cases, though, a component's use of a dependency is isolated from other components. Consider a component that uses a regular expression library to manipulate strings internally. If this component requires a different version of the library than another component does, it should in principle be safe to load both versions since the usage of each library is isolated. Unfortunately, identifying whether two conflicting dependencies are truly isolated from each other is difficult.

Consider the code fragments shown in Figures 7a and 7b. Imagine `pkg.B` extends a previously published class `pkg.C`. Whether the method in `B` overrides `C.m(A)` depends on how `A` is resolved. If `A` resolves to the same class used by `C`, then `B`'s method overrides `m`, otherwise it doesn't. Interestingly, both cases result in fully type-checked code. Figure 7c shows the latter case, where `A` is resolved differently by `B`'s codebase, `CB2`, and Figure 7d shows the dependency graph induced by the two class definitions.

Allowing programs with dependency graphs such as Figure 7d can result in subtle and surprising behavior. For instance, calling the method `m` on an object of type `B` and passing in a parameter of type `pkg.A` from `CB2` will be dispatched differently depending on the type of the *reference* to the object receiving the call. For instance, if the reference has type `C`, then `C.m(Object)` will be called since `C.m(A)` uses the version of `pkg.A` from `CB1`. While it is possible the code's author intended this behavior, it is far more likely that this behavior is unintentional.

Since errors related to inconsistent linkage are difficult to detect and debug, we enforce a constraint on the static dependencies of mobile code. The constraint is most easily expressed in terms of a dependency graph such as the one shown in Figure 7d. Nodes in the graph are published class objects, and each edge is the resolution of a dependency using the source node's codebase. We require that a unique implementation of a dependency be reachable in the dependency graph. In Figure 7d, two implementations are reachable for `pkg.A`, so the compiler would reject publication of `pkg.B`.

Namespace consistency encourages modular design without imposing a specific module system, thereby permitting a wider range of workflows than previous systems. For instance, independently linking components via common interfaces isolates each component's namespace and abstract dependencies from their implementation. Provided the namespaces of these components are consistent, an updated version of one component may link against the other classes without changing them. Programs can evolve incrementally and securely, while avoiding unnecessary re-publication of classes.

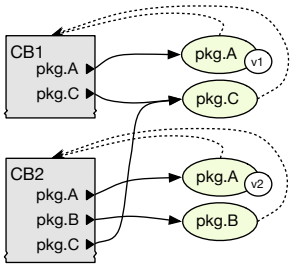
The consistency constraint applies to the *static* dependencies of a class and does not constrain the dynamic type of objects beyond normal type safety. At run time, a reference may point to an object whose class type is neither in the class's codebase nor consistent with its namespace. There is no possibility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

```
1 package pkg;
2 class C {
3   void m(Object o) {
4     ...
5   }
6   void m(A a) {
7     ...
8   }
9 }
```

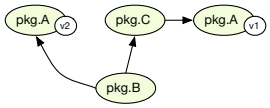
(a) A mobile superclass

```
1 package pkg;
2 class B extends C {
3   void m(A a) {
4     ...
5   }
6 }
```

(b) A subclass of `pkg.C`



(c) Codebases



(d) Dependency graph

Fig. 7: Two classes using different versions of `pkg.A`

6.3. Explicit Codebases

We introduce *explicit codebases* as a language mechanism for referencing specific implementations of dependencies. An explicit codebase is a name for a codebase object published in Fabric. A programmer may use this alias to qualify dependencies that should be resolved through the specified codebase rather than the class's codebase. Explicit codebases may appear at the root of any fully qualified type name. When a name is qualified via an explicit codebase, the namespace of the specified dependency is isolated from that of the dependent class. Because the programmer's intention is unambiguous, dependencies linked via explicit codebases are exempt from namespace-consistency checks.

Explicit codebases may refer to multiple implementations of a dependency in the same namespace. For instance, to override the superclass method `C.m(A)` in `pkg.B`, the class definition should read:

```
1 package pkg;
2 codebase cb1;
3 class B extends C {
4   void m(cb1.A a) {
5     ...
6   }
7 }
```

The publisher associates the alias "cb1" with a Fabric reference to codebase `CB1` on the command line.

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 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 affect program functionality, it may be desirable to isolate the namespace of each component using an explicit codebase.

7. Implementation

To evaluate our design, we extended the Fabric prototype with support for mobile code, as described in this section.

7.1. Compiler

We extended the Fabric compiler to support the new language features and analyzes. Additional extensions were required to enable the compiler to load dependencies from Fabric. Provider-bounded label checking is implemented as part of Jif and is inherited by Fabric. These compiler changes comprise about 12k lines of Java code.

7.2. Class loading

To load and run mobile code, we implemented a Java class loader that is used by worker nodes. When the JVM requests a new class, the class loader fetches the corresponding Fabric `FClass` object, which contains references to its home codebase, as well as the source code. To specify which codebase should be used, the compiler mangles class names mentioned by mobile code to include their home codebases.

After fetching a class object, the 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 label of the code, and the codebase, are also passed to the compiler. The compiler generates Java bytecode and caches it locally in memory and on disk, so that compilation can be reused. After compilation, the loader reads the bytecode from cache and uses the Java class loader API to load it into the JVM.

For bootstrapping purposes, certain system classes are treated specially. They are loaded from bytecode on disk, in much the same way as by the default Java classloader.

The definitions of the `Codebase` and `FClass` classes were written in Fabric and contain about 90 lines of code. The remaining changes to the runtime system comprise about 2,200 lines of Java code and 420 lines of Fabric code.

7.3. Limitations

Some mechanisms are not implemented in the current Fabric implementation. These mechanisms should not have any significant effect on the results reported here.

- Access labels are implemented, but full support for run-time enforcement of access labels in multiworker transactions is incomplete.
- 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 take into account parameters of parameterized types.
- We have not implemented the class object replication scheme of Section 5.6.

```
1 Map {resLbl}
2 createMap (User user, label resLbl, label friendAccess)
3 where
4   { provider ⊔ MapServer.provider ⊔ pc ⊔ {⊥→;⊤←user} }
5     ⊑ { resLbl ⊓ {⊤→user.sn} ⊓ {⊤→ms} ⊓ friendAccess },
6   resLbl ⊑ {⊤→n},
7   localStore ⋡ resLbl
8 {
9   label fetchLabel = {resLbl ⊓ {⊤→ms}};
10  Box   boundary   = new Box(0,0,0,0);
11  for (User friend : user.friends)
12    if (friendAccess           ⊑ {⊤→friend.sn})
13    && {friend→friend.locGrp} ⊑ fetchLabel)
14      boundary.expand(friend.location);
15  Map map = ms.getMap(boundary)
16              .copy(resLbl, localStore);
17  for (User friend : user.friends)
18    if (friendAccess           ⊑ {⊤→friend.sn}
19    && {friend→friend.locGrp} ⊑ resLbl)
20      addPin(annotated, friend.location, friend);
21  return map;
22 }
```

Fig. 8: An important part of the FriendMap code. Some details have been changed for clarity (e.g., Fabric does not currently support Java 5's `for`-loop syntax, and some error handling has been elided).

8. Evaluation

Our architecture has three key goals. First, it should be secure. This topic has been discussed throughout the paper. Second, it should be expressive; it should enable a range of useful applications to be built. Third, it should have acceptable performance.

To evaluate the system, we implemented two example applications, which include the FriendMap application. These examples cover many of the current uses of mobile code, but also enable new functionality. These example applications work correctly on our prototype implementation, demonstrating that provider-bounded security verification allows interesting code and that codebases enable incremental development. Performance measurements from these examples suggest that run-time overhead is acceptable for many uses.

8.1. FriendMap example

To show that Mobile Fabric is sufficiently powerful to securely implement interesting functionality, we implemented a prototype of the FriendMap example. It contains roughly 2500 lines of Fabric code, roughly 200 of which implement the extended versions of FriendMap and Snapp. FriendMap was developed over the course of six weeks by two developers.

As described in Section 2, 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 method `createMap`, which provides the key functionality of FriendMap. This method computes a bounding box of a user's friends (lines 9–14), uses that bounding box to fetch an image from MapServ and construct a private copy (lines 15–16), and then annotates that map with the user's friends' locations (lines 17–20).

The method takes the dynamic labels `resultLbl` and `friendAccess` as arguments. The `resultLbl` argument describes the policy on the created map; it is used in lines 13 and 19 to ensure that friends with private locations will not affect the resulting map.

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, while preventing the user from fetching those objects if the friends' social networks are not trusted to learn about the state of the computation (lines 12 and 18).

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 3–7, which are required to be checked by any method that calls `createMap`.

For example, the first clause (on lines 4–5) guards the flows of information from the code itself (labeled `provider` and `MapServer.provider`), from the fact that the method was called (labeled `pc`), and from the user's set of friends (labeled $\{\bot\rightarrow;\ \top\leftarrow\texttt{user}\}$) to effects on the resulting map (labeled `resLbl`), as well as fetches of the user's object, the map server's initial map, and the friends (with access labels $\{\top\rightarrow\texttt{user.sn}\}$, $\{\top\rightarrow\texttt{ms}\}$ and `friendAccess` respectively).

Omitting any of the `where` clauses or the dynamic checks in this example would lead to exploitable information flows in the FriendMap application. Mobile 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 them.

We also implemented one of the evolution scenarios described in Section 6. We implemented a second version of the Snapp codebase that adds 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 subsequently extended the FriendMap application to make use of this extended functionality. FriendMap version 2 extends FriendMap version 1, and overrides the implementation of the `addPin` method to color the added pin using the user's mood. Because version 2 is a backwards 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.

8.2. Bidding agent example

In this example, a user supplies an agent to choose between two ticket offers made by 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 is about 570 lines of code.

Four parties participate: a trusted broker, two airlines, and the user. They are represented by Fabric principals `Broker`,

```
1 interface UserAgent[label L] {
2   int {L} choice( Offer[L]{L} offer1,
3                   Offer[L]{L} offer2 );
4 }
5 interface Agent[principal A, label L] {
6   void prepareForAuction{A→;A←}();
7   Offer[L]{L} makeOffer {L} (
8     UserAgent[L]{L} userChoice,
9     Offer[L]{L} bestOffer);
10    ...
11 }
```

Fig. 9: Interfaces provided by Broker

```
1 label{Broker←} auction =
2   new label{Broker→;User←;AirlineA←;AirlineB←}
3 Agent[AirlineA,auction] agentA = a.getAgent(auction);
4 Agent[AirlineB,auction] agentB = b.getAgent(auction);
5 UserAgent[auction] userAgent = u.getAgent(auction);
```

Fig. 10: Initializing airline and user agents

AirlineA, AirlineB, and User. Principal Broker is trusted by others: Broker≽AirlineA, Broker≽AirlineB, and Broker≽User; no other trust relationships are assumed. Every principal is associated with a Fabric store.

To facilitate interaction of different mobile agents, Broker publishes interfaces, illustrated in Figure 9, for the airlines' and user's agents. The interfaces use principal and label parameterization, a Fabric language feature 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 uses 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 information about calling this method to be observed by airline A. This permits airline agents to fetch new information from airline airlines, such as seat availability or current lowest prices. Method makeOffer is called during the bidding phase and generates a new offer to the user's agent. The signature of this method records the key feature of our mobile-code framework: the user's agent is passed in as a method argument, 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 10 shows initialization of mobile agents. Lines 1–2 declare a label auction at which offers are produced. The confidentiality component of this label, {Broker→}, records

	Execution time (ms)	
	FriendMap	Bidding
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. 11: Execution time for the steps required to verify and load dynamically compiled mobile code, averaged over five runs.

that an offer may only be read by Broker; the integrity component of this label User←;AirlineA←;AirlineB← record that the choice of an offer may be influenced by all three principals (the user and both airlines). Lines 3–5 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 Broker; for example, to declassify the winner, the broker performs:

```
1 declassify (
2   endorse (winner, {*auction} to {Broker→;Broker←})
3   to {Broker→User;Broker←})
```

Before declassifying winner, this code endorses it to integrity {Broker←}. Without this endorsement, the declassification would not be robust: a potentially untrusted principal User could influence what confidential information he or she learns. Direct endorsement recognizes this influence. Declassification of the winning offer is similar.

8.3. Performance evaluation

Support for mobile code affects the performance of the system in two ways. First, there is additional work required to dynamically load and analyze new code. Second, linking with remote classes imposes some execution overhead.

To evaluate these impacts, we have broken down the execution time of the two examples. These measurements were performed on an Intel Core i7-860 with 4 GiB RAM. Figure 11 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 to be able to reduce that time significantly. More importantly, we can often avoid the analysis entirely—when either the worker has compiled the class in the past, or the worker trusts the provider of the code to correctly compile it.

To demonstrate this, Figure 12 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 back-ported the bidding agent example to the original non-mobile Fabric prototype, and give the load time for that as well.

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

Fig. 12: Total time spent in the class loader, under different conditions. We present the mean over five runs.

	Total execution time (ms)		
	OO7	FriendMap	Bidding
Uncached	—	17,995	14,210
Cached	39	4,910	5,999
Non-mobile	36	—	4,873

Fig. 13: Running time of OO7 and the mobile-code examples

To evaluate the run-time overhead of our codebase mechanism, we reran two traversals from the Fabric implementation of the OO7 Object Oriented Database benchmark suite [6]. This benchmark fetches and calls methods on a large number of objects. To focus on the run-time overhead of mobile code, we ran the benchmarks with compiled bytecode cached locally, and compared this with the original Fabric system. The results of this benchmark are shown in Figure 13.

Comparing Figures 12 and 13 shows that the uncached execution time with class loading removed is less than the cached execution time. This shows that compiling code has side effects, such as populating the object cache, that would have been performed anyway.

9. Related work

The Fabric system [20, 19] supports secure programming in a decentralized system, but it does not have mechanisms for secure mobile code. This paper has shown how to extend the Fabric run-time system and language to support mobile code, and has controlled some covert channels in Fabric.

The DStar system [38] provides OS-level enforcement of secure information flow across a federated system. It uses "exporters" to map machine-level security policies into a distributed context, allowing hosts to define the degree of trust between host nodes. DStar has no notion of code integrity or secrecy, and does not support mobile code; publishing and installation of code lies outside the system. The dynamic enforcement in DStar and in earlier OS-level systems [37, 16] tracks information flows coarsely; to manage information of different security levels, the application must be partitioned accordingly. Declassification is needed more often, and its scope is an entire process rather than a single data item.

Information-flow control for JavaScript has been explored, in which loaded code is dynamically checked against statically identified residual information-flow requirements [7].

Cross-origin resource sharing (CORS) [33] 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. Policies are specified by web sites and users have no explicit control over what policies are enforced.

Various attempts have been made to strengthen isolation guarantees for JavaScript. Conscript [21] applies aspects to JavaScript primitives, isolating loaded scripts in useful ways.

Caja [22] uses capabilities to provide isolation in web mashups. Caja capabilities protect access to resources at a fine granularity. Secure information flow can be enforced by checking capabilities at statically predetermined locations [4], assuming a static analysis of information flow.

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

SPIN [3] is an extensible operating system that allows core kernel functionality to be dynamically specialized by modules written in Modula-3. Like Mobile Fabric, SPIN leverages language-level features like interfaces and type safety to provide isolation for untrusted system modules. Unlike Mobile 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 Mobile Fabric is mostly 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 [10] and Knit [28] 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 [10]) or user-defined constraints (in [28]). 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 [27]. 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.

10. Conclusions

This paper describes a new kind of computing platform: a decentralized platform for running mobile code securely, subject to explicit policies for confidentiality and integrity. A prototype of this platform has been implemented and evaluated on various distributed applications. The platform enables applications that would be disallowed by isolation-based security mechanisms; the explicit policies used to develop the applications help guide secure design.

The new mobile-code architecture is an interesting and potentially useful artifact in its own right; because it addresses a general problem, the principles and techniques that we have described should be useful for making other distributed systems secure, especially those employing mobile code.

Acknowledgments

We thank Danfeng Zhang, Dan Ports, David Schulze, and Barbara Liskov for their suggestions regarding this work and its presentation. Lucas Waye helped implement the class-loading mechanisms of Mobile Fabric. Danfeng Zhang improved declassification and error reporting. This work was funded by a grant from the Office of Naval Research (ONR N000140910652), by two grants from the NSF: 0424422 (the TRUST center), and 0964409. This research is sponsored by the Air Force Research Laboratory.

References

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, October 2008.

[2] Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), September 2011.

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM Symp. on Operating System Principles (SOSP)*, pages 267–284, December 1995.

[4] A. Birgisson and A. Sabelfeld. Capabilities for information flow. In *Proc. 6th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, June 2011.

[5] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 217–230, December 1993.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[7] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2009.

[8] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[9] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[10] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, May 1998.

[11] E. Hammer-Lahav. The OAuth 2.0 authorization protocol. Network Working Group Internet-Draft, September 2011.

[12] Ian Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML, version 1.5446, November 2011. W3C editor's draft, http://dev.w3.org/html5/spec.

[13] 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, April 2002.

[14] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW '07*, May 2007.

[15] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *CCS '07*, October 2007.

[16] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[17] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[18] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[19] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, Owen Arden, Danfeng Zhang, and Andrew C. Myers. Fabric 0.1. Software release, `http://www.cs.cornell.edu/projects/fabric`, September 2010.

[20] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew 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.

[21] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proc. IEEE Symposium on Security and Privacy*, May 2010.

[22] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008.

[23] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[24] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997.

[25] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[26] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[27] Oracle Corp. JAR file specification, 1999. http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html.

[28] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. 4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 347–360, October 2000.

[29] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[30] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[31] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[32] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[33] Anne van Kesteren. Cross-origin resource sharing. W3C working draft, W3C, March 2009. http://www.w3.org/TR/2009/WD-cors-20090317/.

[34] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 203–216, December 1993.

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

[36] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[37] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[38] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proc. 5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

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

[40] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.