Poster: Contract Verification for Mobile Security

Hannah Gommerstadt
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
Email: hgommers@cs.cmu.edu
Graduate Student

Frank Pfenning
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
Email:fp@cs.cmu.edu
Faculty

Limin Jia
Electrical & Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA
Email: liminjia@cmu.edu
Faculty

I. Problem and Motivation

Every second of every day large amounts of confidential data are secured to enable financial transactions, intelligence operations, and countless other tasks. Today, many systems are protected by distributed security mechanisms, such as cryptographic protocols, or reference monitors. These systems consist of components that are connected, but may be located in diverse physically-distant environments. One of the characteristics of a distributed system is its lack of a single point of failure – each connected node poses a unique opportunity for exploitation. The components collaborate to complete joint tasks, such as implementing a cryptographic protocol. Each one of these components has a prescribed role, their *contract*, that governs their behavior during the process of the joint computation. Guaranteeing the security of these systems is a challenge because it is necessary to ensure that every component is not deviating from its expected behavior. A violation of even a single contract can derail the joint computation and cause a serious security breach in the distributed security protocol. In an adversarial distributed computing environment, program components do not trust each other and cannot be trusted to adhere to their contracts. In order to ensure the security and reliability of the system, it is necessary to identify the contract violation and pinpoint the rogue component. This work strives to provide logic and language-based methods to detect contract deviance in distributed systems. More specifically, we define a model for session-typed communicating processes, show how to *dynamically monitor* communication to enforce adherance to session types and present a system of *blame assignment* in case an alarm is raised.

II. Background & Previous Work

In functional languages, a contract for a function can be modeled as an expressive type that places constraints on its arguments and return value. For example, a possible contract for the division function could be:

$$div : x : int \to \{y : int | y \neq 0\} \to float$$

In order to model concurrent computation in distributed systems, we use a session type system which was designed by Toninho et al. Session types are based on a computational interpretation of linear logic which is a substructural logic that allows reasoning about resources within the logic itself. Toninho et al. used session types as the basis for SILL which is a programming language which integrates ordinary functional computation with message-passing concurrent computation [1]. The session type system allows us to reason about concurrent processes that communicate over channels by message-passing. Recently, SILL has been extended to support both synchronous and asynchronous communication by using the logical concept of polarization [2]. We model a distributed system by joining many processes executing in parallel in a synchronous or asynchronous setting.

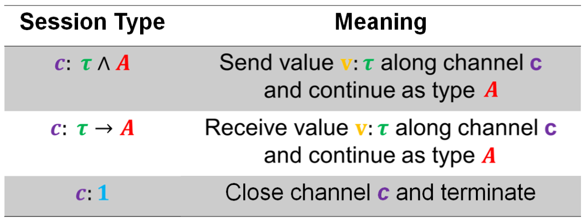
In the presence of message passing concurrency, there are two main reasons why *dynamic monitoring* of communication is necessary. First, when a new process is spawned, part of the execution of a program now escapes immediate control of the original process. If the newly spawned process is compromised, the messages can wreak havoc on the original process. Second, session types allow us to abstract away from local computation because the session type system governs the communication among these processes and enforces a communication protocol. This allows us to safely connect communicating processes written in different languages as long as they dynamically adhere to the session protocol. Because we do not trust remote processes or have access to the code running on each component, dynamic monitoring is a necessity. While there is a significant body of work on the static checking of contracts, there is very little work on dynamic contract checking, especially in a distributed setting.

III. Model

We develop a distributed type-directed monitoring infrastructure that provably detects contract deviation. In our system, every channel has two endpoints – a provider and a client. A process always provides along a single channel, but it may be the client of multiple channels. A session type A prescribes the service provided by process P along a channel c. We use session types as contracts for processes. Figure 1 presents some examples of session types and their computational interpretations.

We assume that all the processes are untrusted, but that the message queues are trusted. The message queues serve as monitors which observe channels that processes communicate over and verify that the correct communication protocol is being observed. Essentially, the monitor acts as a type checker, ensuring that the processes are consistent with the type of the channel they are communicating over. If the monitor detects a typing error, an alarm is raised.

To ascertain that our monitoring infrastructure is adequate and correctly enforces behavior contracts, we prove two cor-



Session Type	Meaning
c: $\tau \wedge A$	Send value v: τ along channel c and continue as type A
c: $\tau \to A$	Receive value v: τ along channel c and continue as type A
c: 1	Close channel c and terminate

Fig. 1. Computational interpretations of session types

rectness properties. We first prove that no process is considered deviant, the monitor is unobservable. That is, the existence of a monitor does not alter the observable computation of the system. We also prove that when an alarm is raised by a monitor, the deviation is attributed to a set of potential processes where the breach could have originated.

IV. Example

We consider a camera application for the Android platform that takes photos using the phone's built in camera. When the user downloads the application to the phone, the application requests the permissions it requires to function, such as access to the camera. However, we might like the camera application to seek permission from the user every time it takes a picture to avoid the situation of the camera constantly taking photos which may potentially cause a security violation.

This example is modelled by three processes: the Android operating system, the user and the camera application. Using our model, we can concisely encode the behavior contracts of the camera application and the user as session type declarations.

$$\begin{aligned} &stype\ Cam = \&\{take : photoPerm \multimap picHandle \otimes Cam\} \\ &stype\ User = \&\{picPerm : \\ &\qquad\qquad \oplus\{fail : User; succ : photoPerm \otimes User\}\} \end{aligned}$$

The Cam type, which models the behavior of the camera application, states that the camera offers one service, the function $take$. This function requires the input of a photo permission that it uses to produce a handle to a picture and then it continues behaving as the camera. The $\&$ denotes an external choice that allows any application to request any service Cam offers, which in this case is only $take$.

The $User$ type, which models the user's behavior, also offers one service $photoperm$. This function allows to user to deny the photo permission ($fail$) or to approve the permission ($succ$). As before, the $\&$ denotes an external choice that allows any application to request any service $User$ offers, which in this case is only $photoperm$. The $\oplus$ denotes an internal choice coming from the user that allows the user to choose to either return $succ$ or $fail$.

We note that the types specify the desired communication patterns, not the properties of the values being send across channels. We also note that this example is indicative of more complicated constraints we can impose on the use of sensitive data on mobile phones such as location data coming from

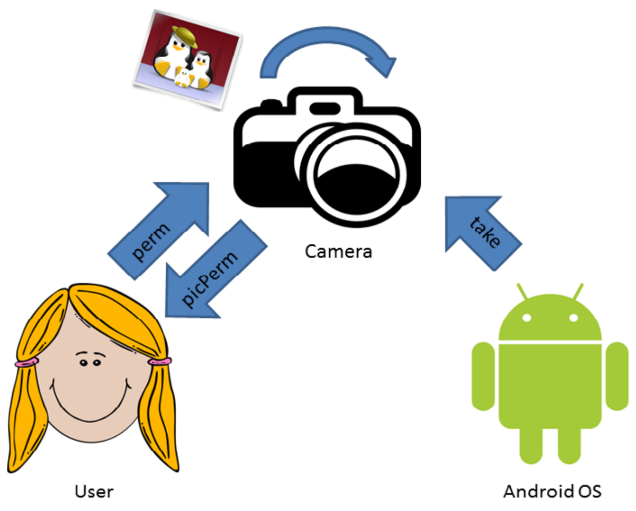


Fig. 2. Interaction between the Android OS, the camera and the user.

the GPS. The following is a snippet of code that models the interaction between the camera and the user. We have a camera application called cam which has type Cam and a user called usr with type $User$.

```
send cam take;
send user picPerm;
case user of
         fail =>
         succ => perm ← recv user;
                 send cam perm;
                 pic ← recv cam;
```

Fig. 2 provides a visual interpretation of the above code. First, the Android operating system asks the camera to take a picture, then the camera requests a picture permission $picPerm$ from the user. The user decides whether she would like to approve the permission request. If she does not approve it, nothing happens. If she approves it, she sends the permission $perm$ to the camera. The camera can then produce a handle to a photograph pic.

References

[1] F.Pfenning and D.Griffith. Polarized substructural session types. In A. Pitts, editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS) 2015*, London, England. Apr. 2015. Springer LNCS. Invited Talk. To appear.

[2] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions and sessions: A monadic integration. In M. Felleisen and P. Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP 2013)*, pages 350-369, Rome, Italy, Mar. 2013. Springer LNCS 7792.