The Future Mechanism and Information Flow Security

Farzane Karami, Christian Johansen, Olaf Owe and Gerardo Schneider

September 12, 2018

The Future Mechanism and Information Flow Security

Farzane Karami[a], Christian Johansen[a], Olaf Owe[a], Gerardo Schneider[b]

[a]*Department of Informatics, University of Oslo*
[b]*Department of Computer Science and Eng., Chalmers University of Technology*

Introduction

Security for distributed systems is a critical issue since a large number of users and systems are affected by such systems. Challenges of information flow analysis of distributed systems depend on the communication paradigms used and their semantics. The *"actor paradigm"* [1] is a model advocated for designing distributed systems, in particular, it offers modular and compositional design and analysis. In addition to the actor model, the object-oriented paradigm has become popular because of its facilities for program structuring and reuse of code. These two paradigms are combined in the so-called *"active object"* model, where the objects are concurrent and autonomous, communicating with other objects by "asynchronous methods" [3].

We briefly discuss communication paradigms in active object languages using the syntax of the ABS (abstract behavioral specification) language [5]. A synchronous and blocking call of method m on a remote object o has the form $x := o.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. The caller is blocked until the callee returns the value, leading to unnecessary waiting. One way of avoiding blocking is achieved by using futures proposed in [2] and exploited in MultiLisp [4], ABCL [8], and several other languages [3]. A future is a read-only placeholder which eventually will contain the return value from an asynchronous method call [4, 7]. When a remote method call is made, a future object with a unique identity is created. The caller may continue with other computations while the callee is computing the return value. When the return value is computed, it is stored in the future object. The future is then said to be *resolved*. In a call statement $f := o!m(\overline{e})$, f is a *future variable* used to hold the future identity of the call, and the symbol "!" indicates an asynchronous method call. In the case of first-class futures, a future identity can be passed to objects desiring the return value of the corresponding method call, even before the value is computed. Thus, a future can be distributed to many active objects in a system.

The prevalence of futures in active object languages [3] highlights the significance of investigating inherent security and privacy issues related to futures. Futures might contain highly sensitive data, while in the case of first-class futures, any object that has a reference to it can access the content. Inside an

Email addresses: farzank@ifi.uio.no (Farzane Karami), cristi@ifi.uio.no (Christian Johansen), olaf@ifi.uio.no (Olaf Owe), gerardo@cse.gu.se (Gerardo Schneider)

object, high-sensitive data might be leaked from futures to low-secure variables, and be transmitted to low-security actors, observable by an attacker. Consequently, it is critical to track futures and analyze their information flow security.

Information flow security challenges regarding futures

Future variables give a level of indirectness in that the retrieval of the result of a call is no longer syntactically connected to the call, compared to future-free languages. For instance when the future is received as a parameter, it may not statically correspond to a unique call statement. One may overestimate the set of call statements that correspond to this given future parameter, but it requires access to the whole program. Therefore, when allowing futures as parameters, static information flow analysis would be imprecise, because the set of external calls that may result in an actual parameter is not statically given, and these calls are not uniform with respect to secrecy levels. In this case, one must consider the worst case possibility (i.e., the highest secrecy level) for the set of possible corresponding call statements, which is too conservative and severely limits statically acceptable information passing and call-based interaction, or requires dynamic checking.

In addition, the future concept comes with a notion of future identity, but not a notion of associated caller, callee. Identities of the caller and callee could in principle be incorporated in the future identity, but only at run-time. At static time there is no information about the caller and the creator of a future. This opens up for third party information with indirect/implicit handling of sensitive information. Static information flow regarding futures is too conservative. It causes unnecessary rejection of programs, especially when the complete program is not statically known as is usually the case in distributed systems. This is not desirable for To address this problem, we propose an approach based on the notion of "*wrappers*" [6]. By wrapping futures and all objects receiving futures (recipient objects) as parameters or return values, we can prevent leakage of information from futures with high-sensitive data at the cost of dynamic checking.

We statically identify futures and recipient objects in the program. Then at dynamic time wrappers around them monitor and control communicated messages to or from these objects. When a low security object attempts to access a high secured future, the access should be rejected because of incompatible secrecy levels.

The idea of wrappers is a permissive and precise dynamic approach, using the run-time environment to track information flow and monitoring the execution inside an object to prevent security violations. We modify the run-time environment, i.e., add another component to the environment to keep the secrecy levels of variables and change the operational semantic rules in a way to track flow-sensitive information flow dynamically. Fig. 1 exemplifies information flow analysis regarding futures and wrappers. An asynchronous method call toward object O creates future f, arrows 1, 2. Then it is passed to object P as a parameter of a method call. The future f, object O', and the recipient

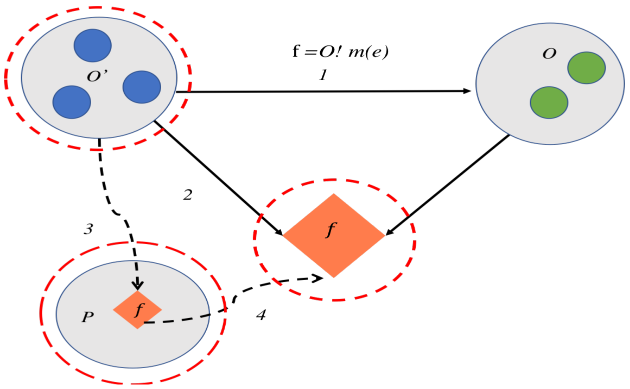

Figure 1: Information flow security regarding wrappers.

object P are wrapped by wrappers, marked in red, which checks whether it is safe to let them get the future value.

Conclusion

Futures are invented as a flexible way for sharing results and communication; however, their security and privacy are problematic. The notion of wrappers has been developed for safety of objects [6]. We here exploit wrappers for dealing with information security, by extending the runtime system with secrecy levels and applying dynamic checking for securing the use of futures.

References

[1] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MIT Press Cambridge, MA, USA, 1986.

[2] H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.

[3] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, et al. A survey of active object languages. *ACM Computing Surveys (CSUR)*, 50(5):76, 2017.

[4] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[5] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2011.

[6] O. Owe and G. Schneider. Wrap your objects safely. *Electronic Notes in Theoretical Computer Science*, 253(1):127–143, 2009.

[7] Y. Yokote and M. Tokoro. Concurrent programming in concurrent SmallTalk. In *Object-oriented concurrent programming*, pages 129–158. MIT Press, 1987.

[8] A. Yonezawa, editor. *ABCL: An Object-oriented Concurrent System.* MIT Press, Cambridge, MA, USA, 1990.