Securing Mobile Agents Control Flow using Opaque Predicates

Anirban Majumdar and Clark Thomborson

Department of Computer Science, The University of Auckland.
Private Bag 92019, Auckland, New Zealand.
{anirban, cthombor}@cs.auckland.ac.nz

Abstract. Mobile agent technology is an evolving paradigm that combines the inherent characteristics of intelligent agents, namely, adaptability, reactivity and autonomy with mobility. These characteristics of mobile agents provide an excellent means of meeting the distributed and heterogeneous requirements for many electronic commerce applications involving low bandwidth and intermittently connected networks. However, the lack of security in the form of code confidentiality renders this paradigm unsuitable for commercial software. In this paper, we address the problem of mobile agent security by proposing a novel method of mobile agent obfuscation using the concept of opaque predicates to prevent adversaries from observing the control flow of agent code. We discuss about the efficiency of our proposed methodology by demonstrating that to an adversary, the problem of determining the outcome of such opaque predicates is often intractable.

1 Introduction

In the past few years, mobile agent systems have been brought up with the research and development of distributed computing. However, in spite of its tremendous potential, several technical requirements must be met in order to support the widespread transition of agent technology to the commercial domain. Confidentiality of agent code is of foremost concern. The countermeasures directed toward agent protection are radically different from those used for host protection. Host protection mechanisms are a direct evolution of traditional mechanisms employed by trusted hosts and traditional mechanisms are not devised to address threats originating on agents from the execution environment. Agents executing in electronic commerce applications cannot trust the platforms they are executing on and this problem stems from the inability to effectively extend the trusted environment of an agent's host platform to other agent platforms visited by the agent.

Previous works on provable mobile agent security have proposed cryptographic techniques [2] to protect agents from unauthorised code interception; however, since any information belonging to a mobile agent is completely available to its host system, it cannot possibly keep the cryptographic key secret from the system on which it is running. Moreover, the encrypted agents will become susceptible to attacks by the platform once they are decrypted into executable forms. In this paper,

Preprint version. Authoritative version appears in *Proc. KES 2005*, LNCS 3683, pp. 1065-71, http://dx.doi.org/10.1007/11553939_149.

we address the problem of agent security using obfuscation [1], which is a technique to obscure the agent code in such a way that an adversary will not be able to gain a complete understanding of its function (with respect to specification and data).

Our work focuses on obfuscating agent behaviours by introducing opaque predicates to guard the control flow. We show that for an adversary to detect the outcome of such predicates, static analyses of the agent code have to be successfully performed and this problem is often intractable in the presence of aliased pointers and concurrency. We also outline the typical scenarios in which such pointer analyses will be difficult to perform. Our technique can be used in conjunction to other obfuscation techniques using aliasing.

2 Mobile agent protection issues and related work

The definition of what constitutes an attack depends on what assurances the agent owner needs in order to use a mobile agent. Hohl [3] classified different attack categories that could be mounted on mobile agents by adversaries. We illustrate Hohl's analysis as an attack tree in figure 1. Using our model of protection, we specifically address attacks originating out of spying the control-flow. We achieve this branch confidentiality by obscuring the real control flow of behaviours behind irrelevant statements that do not contribute to the actual computations such that it is impossible for an adversary to find out the correct behaviour from mobile agent code by statically analysing it. An adversary with no semantic understanding of correct control-flow of the code will also find it hard to do purposeful manipulation of the code.

Obfuscation for mobile agent code protection was first addressed by Sanders et al [2] in the form of '*mobile cryptography*'. This technique facilitated development of programs that could operate on encrypted data. However, [3][4] points out that this method is not applicable to generic agent codes since it has the restriction that agents can send cleartext data to only trusted hosts. Hohl [3] extended the concept of "black-box security" by incorporating time-limitedness.

However, his method makes explicit assumption of synchronised global clock for token passing between untrusted servers and is therefore difficult to apply in mobile agent interaction scenario which is inherently distributed in nature. Moreover, his technique failed to correlate between the obfuscation techniques and the corresponding time-limitedness they guaranteed. Sakabe et al's [4] attempt to obfuscate mobile agents using aliasing [6][7] is the only work that provides a theoretical basis for obfuscating mobile agents. Wang et al. [5] first proposed an obfuscation technique based on the difficulty of statically analysing aliased pointers in C programs. They manipulated branch targets using aliased pointers and established that the problem of precisely determining indirect branch targets is NP-hard. Sakabe's obfuscation technique takes advantage of polymorphism and exception handling mechanism of Java and established that the problem of *precisely* determining if an instance of a class points-to an overloaded method during an execution of a program is NP-hard.

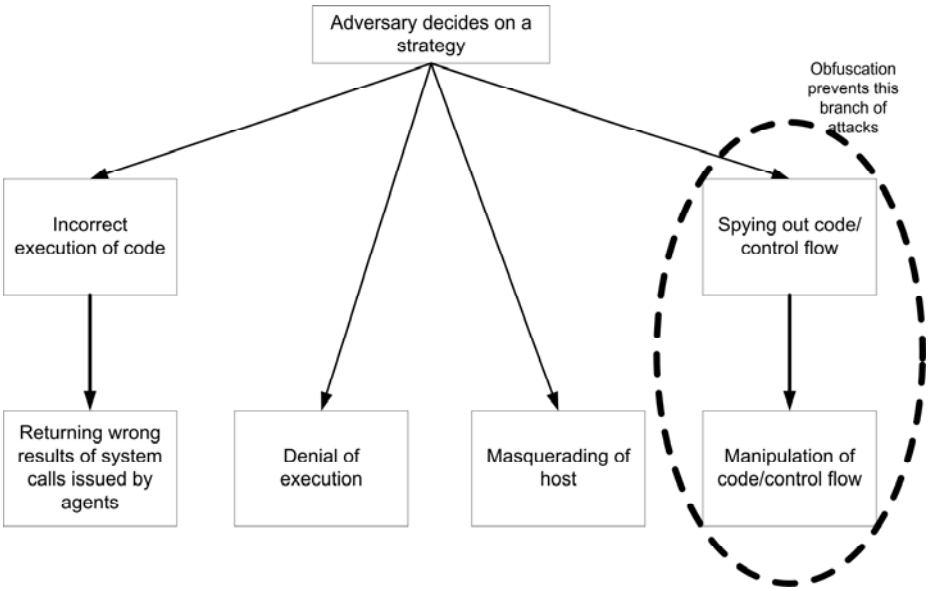

Fig. 1. The "attack tree". Obfuscation using opaque predicates will attempt to prevent attacks marked with the dotted oval from taking place.

3 Use of opaque predicates for mobile agents obfuscation

Obfuscation is the technique of transforming a program into a form that is more difficult to understand for either a human adversary or for an automated one or both, depending upon the transformation applied [1]. An obfuscated program should have "*identical*" behaviours with respect to the original unobfuscated one. However, we relax this stringent form of restriction by allowing the obfuscated program to have side effects. In this section, we focus on a particular obfuscation class which obscures the control-flow of a program using *opaque predicates*.

An opaque predicate is a conditional expression whose value is known to the obfuscator, but is difficult for the adversary to deduce. A predicate P is defined to be *opaque* at a certain program point *p* if its outcome is only known at obfuscation time [1]. We write P^F_p (P^T_p) if predicate P always evaluates to False (True) at program point *p*. The opaqueness of such predicates determines the resilience of control-flow transformations.

Mobile agent frameworks are instances of loosely-coupled message-passing distributed systems which have the property that communications incur latency and computations proceed at different speeds. Thus, agents do not have any predetermined scheduling policy. This intrinsic nature of mobile agents facilitates programmers to incorporate a large amount of concurrency between them.

If the control-flow of agent P is to be obfuscated using opaque predicates, a certain number of *guard* agents belonging to the system are employed to achieve this

task. Since agents in mobile agent systems typically collaborate through message exchanges to achieve a particular task, the set of guard agents could be those that agent P frequently communicates with. The actual number of guard agents employed in protection of a single agent will depend *dynamically* on the availability of agents in the system. It then initialises a data structure such as circular linked-list with some initial pointers pointing on it. Agent P then sends a message, containing this data structure, to one of the guards. Each guard, when it receives the data structure, may change one of its pointers and will then try to send the data structure along to another guard or back to agent P. The message-passing pattern and pointer update by guards must maintain an invariant that holds on the data structure when it is sent back to P. We shall illustrate this protocol by a simple example as depicted in figure 2. Let four guard agents be dynamically spawned by P. We call these guards A, B, C, and D. Agent P initialises a circular linked-list and passes it randomly to a guard which initiates the message-passing between itself and other guards. Let the A be the initial guard agent and it updates pointer *p*. Similarly, guards B, C, and D manipulate pointers *q*, *r*, and *s* respectively.

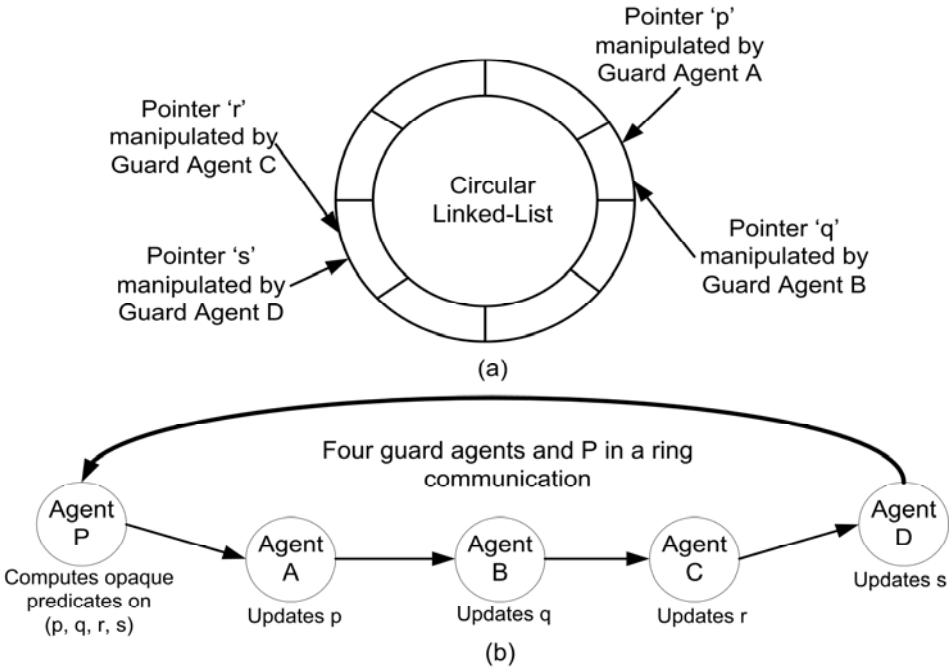

Fig. 2. The global dynamic data structure in the form of a circular linked-list shared by four guard agents A, B, C, and D and a simple pointer update protocol.

During the initialisation process, P also embeds the linked-list update-invariants in each of the guard agents such as:

- *p* and *q* are aliased.
- *r* and *s* are aliased.
- *p* and *q* never alias pointers *r* and *s* and vice-versa.

In figure 2(b), we have illustrated the communication pattern between agent P and guards with a simple ring protocol. In practical systems, however, more complicated communication patterns could be used. After updating its respective pointer, each guard agent passes on the data structure to its successor. Finally, agent P receives the data structure from guard D, after which, it proceeds to construct and check opaque predicates using aliased pointers *p*, *q*, *r*, and *s* as illustrated in the pseudo-code snippet of figure 3.

Guard Agent:

```
…
//initialisation
receive <update_rule, Agent(P)>
//perform update
while (true) {
       receive <data_structure, Agent(ID)>;
       update (pointer);
       send <data_structure, Agent(ID)>;
…
}
```

Obfuscated Agent P:

```
…
initialise (data_structure);
send <data_structure, Agent(A)>;
…

//receive the data structure from Guard Agent D
while (!receive <data_structure, Agent(D)>) {
       wait;
}
//send the data structure for another round of updates
send <data_structure, Agent(A)>;
//initiate testing on pointer invariants

if (data_structure.p == data_structure.r &&
data_structure.q == data_structure.s)
{ // Opaquely-False Predicate
       // perform dummy behaviour
}
else {
       //perform real behaviour
}
if (data_structure.p == data_structure.q &&
data_structure.r == data_structure.s)
{ // Opaquely-True Predicate
       // perform real behaviour
}
else {
       //perform dummy behaviour
}
```

Fig. 3. Pseudo-code showing sample guard agent pointer update action and obfuscation of control-flow in P using the aliased opaque predicates.

The branch obfuscated using opaque predicate [(*p*==*q*) && (*r*==*s*)] only evaluates to true; whereas, branches obfuscated using any other combination evaluate

to false. Before checking for the outcome of opaque predicates, the obfuscated agent P waits for the data structure from guard agent D. After receiving the data structure, it then sends it around for another pointer update round. These data structure passing messages could be tagged with a special value to distinguish from other messages.

In order to statically analyse the obfuscated code, the adversary must depend on a static slicer to find those parts of a program which could affect the value of opaque predicates at points of interest (namely, at locations where behaviours are obfuscated). The slicing of distributed programs is a major challenge due to the timing related interdependencies among processes. Moreover, to find the slicing criterion of the slicer, the analyser must rely on alias analysis [6][7][8] to determine what kind of structure the pointers point to at runtime, and if the pointer corresponding to the variables used in the construction of opaque predicate may/must refer to the same dynamic object in the guard agents at some program location (where the opaque predicates are used in P). Two pointers referencing the same memory location are called *aliases*. Therefore, the static analyser must use inter-process escape alias analysis to determine the objects that can be referenced by pointers in processes other than the processes in which they are allocated.

Though numerous works on intra- as well as inter-procedural alias analysis and inter-procedural thread escape analysis [9][10] have been done throughout the last few decades, we have not come across a method that can perform alias analysis by considering asynchronous message-passing of distributed processes as escape points. We believe the reason why this problem has not yet been addressed by the program analysis community is because we still do not have efficient, precise and scalable algorithms for performing simpler cases of alias analysis in sequential multi-threaded programs and hence are unsure how to go about solving this problem which is much more complex in nature.

4 Conclusion

In this paper, we have addressed the problem of mobile agent code obfuscation using opaque predicates, which are structures inserted at program control points to obfuscate the branching of agent behaviours. Obfuscation using opaque predicates that use the inherent concurrency associated with mobile agent systems and aliasing are resilient against well known static analysis attacks. We have demonstrated that it will be very difficult for an adversary to understand the dynamic structure of the predicate and mount attacks that could statically analyse the associated values for each of the terms present in the predicate. The predicate structure can also be made arbitrarily complex by incorporating numerous guard agents dynamically and this demonstrates the flexibility of our technique. Moreover, in an already existing message-passing scenario between agents, the messages exchanged between the obfuscated agents and the guards will contribute to a negligible amount of extra overhead. The technique also does not depend of any particular language feature and is therefore applicable to generic mobile agent platforms.

We note that our technique is not an alternative to the one proposed by Sakabe et al. Obfuscation using method aliasing as a standalone technique may not be

sufficiently resilient since the smaller size of agent programs compared to that of commercial Java software may result in a fewer number of methods to be considered for overloading in Sakabe's technique. Hence, using our technique in conjunction to the one already proposed by Sakabe et al will substantially strengthen the resilience of obfuscated agents.

In this contribution, we make the assumption that an adversary is only able to statically manipulate an agent executing on a platform but not its guards. By imposing this restriction, we prevent the adversary from getting a complete understanding of data structure passing and pointer update rules. Future work will be concentrated on investigating the classes of resilient opaque predicates that demonstrate provable resistance against well known static analysis attacks as well as dynamic analysis attacks including dynamic message interception attacks.

References

1. Collberg, C., Thomborson, C., and Low, D.: *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.* In Proceedings of 1998 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). 1998.
2. Sander, T., and Tschudin, C.F.: *Protecting mobile agents against malicious hosts.* In Vigna G., ed.: Mobile Agents and Security. Volume 1419 of Lecture Notes in Computer Science. Springer-Verlag. 1998
3. Hohl, F.: *Time limited blackbox security: Protecting mobile agents from malicious hosts.* In Vigna G., ed.: Mobile Agents and Security. Volume 1419 of Lecture Notes in Computer Science. Springer-Verlag. 1998
4. Sakabe, Y., Masakazu, S., and Miyaji, A.: *Java obfuscation with a theoretical basis for building secure mobile agents*: . In Lioy A., Mazzocchi D. eds.: Communications and Multimedia Security (CMS 2003). Volume 2828 of Lecture Notes in Computer Science. Springer-Verlag. 2003.
5. Wang, C., Hill, J., Knight, J.C., and Davidson, J.W.: *Protection of software-based survivability mechanisms*. In Proceedings of the 2001 conference on Dependable Systems and Networks. IEEE Computer Society. 2001.
6. Horwitz, S.: *Precise Flow-insensitive may-alias in NP-hard.* ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 19 No. 1, 1997.
7. Hind, M., Burke, M., Carini, P., and Choi, J.D.: *Interprocedural pointer alias analysis.* ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 21 No. 4, 1999.
8. Rugina, R. and Rinard, M.: *Pointer analysis for multithreaded programs*. In Proceedings of 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99). Atlanta, GA, USA. 1999.
9. Salcianu, A. and Rinard, M.: *Pointer and escape analysis for multithreaded programs*. In Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '01), Snowbird, UT, USA. 2001.
10. Whaley, J. and Rinard, M.: *Compositional pointer and escape analysis for Java programs*. Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, CO, USA. 1999.