Manufacturing Opaque Predicates in Distributed Systems for Code Obfuscation

Anirban Majumdar **Clark Thomborson**

Secure Systems Group, Department of Computer Science
The University of Auckland,
Private Bag 92019, Auckland, New Zealand.
Email: {anirban|cthombor}@cs.auckland.ac.nz

Abstract

Code obfuscation is a relatively new technique of software protection and it works by deterring reverse engineering attempts by malicious users of software. The objective of obfuscation is to make the logic embedded in code incomprehensible to automated program analysis tools used by adversaries. Opaque predicates act as tool for obfuscating control flow logic embedded within code. In this position paper, we address the problem of control-flow code obfuscation of processes executing in distributed computing environments by proposing a novel method of combining the open problems of distributed global state detection with a well-known hard combinatorial problem to manufacture opaque predicates. We name this class of new opaque predicates as *distributed opaque predicates*. We demonstrate our approach with an illustration and provide an extensive security analysis of code obfuscated with *distributed opaque predicates*. We show that our class of opaque predicates is capable of withstanding most known forms of automated static analysis attacks and a restricted class of dynamic analysis attack that could be mounted by adversaries.

Keywords: Code obfuscation, opaque predicates, distributed predicate detection, software protection, mobile code protection, and distributed systems security.

1 Introduction

Software obfuscation is a protection technique for making code unintelligible to automated program comprehension and analysis tools. It works by performing semantic preserving transformations such that the difficulty of automatically extracting computational logic out of the code is increased. The first formal definition of obfuscation was given by Barak *et al.* (2001) where an obfuscator was defined in terms of a compiler that takes a program as input and produces an obfuscated program as output. Two important conditions that need to be preserved while making this transformation are (a) *functionality:* the obfuscated program should have the same functionality (input/output behaviour) as the input program, and (b) *unintelligibility:* the obfuscated program should be unintelligible to the adversary in some sense. Barak *et al.* defined an obfuscation method as a failure if there exists at least one program that cannot be completely obfuscated by this method, that is, if any adversary could learn something from an examination of the obfuscated version of this program that cannot be learned (in roughly the same amount of time) by merely executing this program repeatedly. Their negative result established that every obfuscator will fail to completely obfuscate some programs.

Since Barak's landmark paper on the impossibility of obfuscation, focus has shifted to finding obfuscating transforms that are *difficult* (but not necessarily *impossible*) for an adversary to reverse engineer. The goal of such research is to find sufficiently difficult transforms such that the resources required for undoing them are too expensive to be worth the while of adversaries. Following this line of research, we propose in this contribution, an obfuscation technique derived from the combination of an instance of a hard combinatorial problem and the difficult problem of global state detection in distributed systems.

Depending on the size of software and the complexity of transforms, a human adversary may find the obfuscated code difficult to comprehend. However, as Thomborson *et al.* (2004) noted, software that is simple and manageable enough to be completely analysed by human adversaries could presumably be redeveloped from scratch by attackers at reasonable cost. It is up to the software developer to decide against using complicated obfuscation transforms that might overwhelm the performance of his simple efficient software. We will not address issues related to performance/security tradeoffs in this contribution; nevertheless, the purpose of making such observation at the beginning of this paper is to justify the focus of this paper on an obfuscation method that increases the difficulty of analysing complex programs.

Distributed computing obfuscation could be useful in a number of practical scenarios where it is necessary to maintain code confidentiality. In the first example, consider a distributed electronic commerce bidding scenario where the bidders download seller's code for bidding. The seller's code may contain privileged information such as reserve price and prioritized selection list of bidders (such as frequent bidders may have higher rating than first time bidders). The seller would like to keep such information confidential to the bidders, especially when their programs are executing on hosts owned by bidders, at least for the duration of the auction. Code obfuscation would serve as an appropriate tool in achieving this objective. Secondly, consider a grid computing scenario, like the SETI@home (2005) setup, where scientific computation codes are downloaded on untrusted personal computers connected to the global network of loosely-coupled machines. These machines are owned by users willing to contribute a portion of their machine's processing power and time for helping the project compute a section of its scientific result by executing the downloaded code. Here too, it may

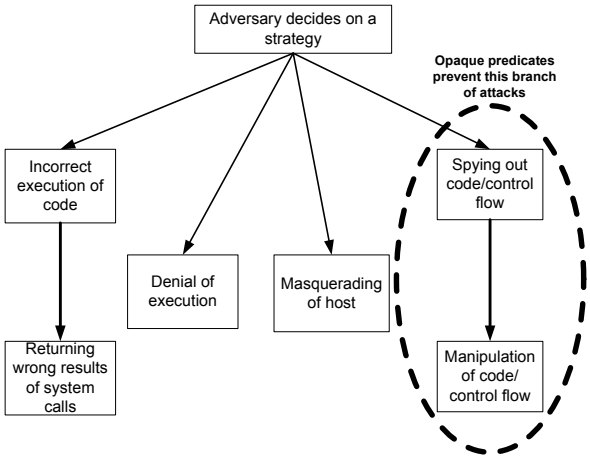

Figure 1: The *attack tree*. The class of attacks marked with the dotted oval are specifically addressed by control-flow obfuscation using opaque predicates.

be desirable that scientific computation logic be kept obscure to the owner of the host. Lastly, distributed obfuscation would be most useful in hiding watermark construction code (Palsberg *et al.* 2000, Nagra & Thomborson 2004) which are used for proving ownership of software. Note that ownership proofs are most important during the economic lifetime of the software product. In all three scenarios, obfuscation need not be perfect in the sense of Barak. Instead, obfuscation is useful if it delays the release of confidential information for a sufficiently long time (Hohl 1998). Secondly, any obfuscation technique would increase the confidence of the code-sender, but might decrease the confidence of a code-executer because it would make it harder to understand what the code is doing.

Control-flow obfuscation by means of opaque predicates was introduced by Collberg *et al.* (1998). An opaque predicate is a construct with true/false outcome. The opaqueness property of predicates is attributed by the fact that though their outcome is known at obfuscation time, it is hard for a deobfuscator to deduce from automated program analysis trace. These constructs are specifically useful for addressing attacks originating out of spying the control-flow as illustrated in the *attack tree* of Figure 1. This branch confidentiality is achieved by obscuring the real control flow of behaviours behind irrelevant statements that do not contribute to the actual computations. 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.

The rest of the paper is structured as follows: In section 2, we introduce notation for discussing distributed opaque predicates. Section 3 illustrates with an example how distributed opaque predicates could be constructed in distributed systems. In section 4, we present a security analysis of our technique. We conclude our paper with a summary and discussion of future work in section 5.

2 The concept of distributed opaque predicates and global states in distributed systems

We define a *distributed opaque predicate* (Φ) as an opaque predicate which depends on local states of multiple processes spread across the distributed system for its evaluation. The activity of each process is modeled as execution of a sequence of events. Communication in distributed systems is accomplished through the communication primitive events *send(m)* and *receive(m)*, where m denotes the message. In asynchronous message-passing systems, information may flow from one event to another either because the two events belong to the same process, and thus may access the same local state, or because the two events are of different processes and they correspond to the exchange of a message.

Without a global clock, events can be ordered only based on the notion of causality which states that two events are constrained to occur in a certain order only if the occurrence of the first may affect the outcome of the second. In distributed systems, we use a happened-before relation, $\rightarrow$ between states to denote this causality (Lamport 1978). The happened-before relation can be formally stated as: $a \rightarrow b$ if and only if: a occurs before b in the same process or the action following a is a send of a message and the action preceding b is a receive of that message. Two states for which the happened-before relation does not hold in either direction are said to be *concurrent*. The concurrency relation $\parallel$, can be formally stated as: $a \parallel b \Rightarrow (a \not\rightarrow b \wedge b \not\rightarrow a)$. A set of states is called a *consistent cut* if all states are pairwise concurrent.

Palsberg *et al.* (2000) defined the concept of *dynamic* opaque predicates as a possible improvement over *static* opaque predicates defined originally by Collberg *et al.* (1998). Their dynamic opaque predicates were constant over a single program run but varied over different program runs. We extend their concept of dynamic opaque predicates by designing distributed opaque predicates to be *temporally unstable*. A temporally unstable distributed opaque predicate can be evaluated at multiple times at different program points $(t_1, t_2, ...)$ during a single program execution such that the values $(v_1, v_2, ...)$ observed to be taken by this predicate are not identical, that is, there exists i, j such that $v_i \neq v_j$. There are a couple of advantages of making distributed opaque predicates temporally unstable. The first one concerns its reusability; one predicate can be reused multiple times to obfuscate different control flows. The second one relates to its resilience against static analysis attacks. As will be explained later in details, distributed opaque predicate values (v_i) depend on predetermined embedded message communication pattern between different processes participating in maintaining the opaque predicate. The communication pattern serves as an invariant for maintaining the consistency of local states updates and these in turn make the predicate go true or false at desired program locations. It is hard for the attacker to statically deduce predicate values because this pattern is:

- distributed over the processes.
- generated on-the-fly only when processes execute.

Structurally, we design distributed opaque predicates to be relational in nature and of the form:

$$\Phi : [(a + b + c + \ldots + n) \ \Re \ K]$$

where $(a, b, c, \ldots, n)$ are integers whose values are set by individual processes (this forms the local state of the process, as explained in the next section), $\Re$ denotes an equality (inequality) operator such as `'='` (`'!='`) and K is a constant. Opaque predicates that are structurally relational are stealthy in the following sense: an adversary who discovers a relational construct in a program cannot conclude with absolute certainty that it is a distributed opaque predicate since common conditional constructs appearing in programs are often relational in nature. But

the most important purpose of making distributed opaque predicates structurally relational lies in the difficulty of detecting this class of predicates in the context of distributed global state monitoring. Relational predicates cannot be written as a Boolean expression of local predicates and therefore presents foremost difficulty in distributed global state detection (Chase & Garg 1995). Our rationale will be further clarified in section 4, where we provide a full security analysis for our class of distributed opaque predicates. A detailed discussion on the difficulty of distributed global state monitoring is outside the scope of this contribution and the reader is encouraged to see Chase & Garg (1995) and the references contained therein.

In the next section, we will illustrate how distributed opaque predicates can be generated from an instance of a hard combinatorial problem in distributed systems. An obfuscator will automatically embed distributed opaque predicates in a distributed systems program and insert *send/receive* primitives for generating a predetermined communication invariant. The communication invariant, in turn, maintains the consistency of local states, that is, the value of each component in the predicate (Φ) so that the predicate holds true (Φ^T) or false (Φ^F) at predetermined control-flows decided by the obfuscator and we will argue that to an attacker, predicate value at every obfuscated control-flow seems unknown ($\Phi^?$).

3 Generation of distributed opaque predicates for distributed systems

We present here different design issues an obfuscator needs to deal with and a step-by-step approach for generating distributed opaque predicates in the context of distributed computing obfuscation.

3.1 Selecting/spawning *guard* processes

Let us assume that a distributed computing system consisting of a set of n inter-communicating processes, denoted by $\{P_1, P_2, P_3, ..., P_n\}$, executes on multiple heterogeneous hosts. Assuming that the control-flow of process P_1 is to be obfuscated using distributed opaque predicates, the obfuscator selects or spawns a certain number of *guard* processes to aid in the obfuscation of P_1. Since processes in distributed systems typically collaborate through message exchanges to achieve a particular task, the set of *guards* could be those processes P_1 frequently communicates with. The actual number of *guards* employed in the obfuscation of a single process may depend dynamically on the availability of processes. However, the obfuscator may spawn dummy processes to serve as *guards* if there are not enough processes in the system to do this task. The basic idea is to distribute the local states formed in the construction of distributed opaque predicate in P_1 amongst the *guards* and embed a communication pattern in the form of *send/receive* calls that will update respective local states of processes to previously known values. The local state update rules and communication pattern embedding are described in the following subsections.

We illustrate the process interaction architecture in Figure 2. For the demonstration to follow, we have selected two processes, P_2 and P_3, to serve as *guards* for P_1. Local state for each process i is denoted by the variable $p_i.v$. P_1 could, in turn, serve as a *guard* process for helping in obfuscating any other process within the system but we have excluded that possibility for the sake of keeping this illustration simple.

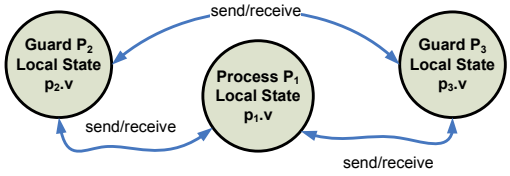

Figure 2: The protected process P_1 with local state $p_1.v$ and two *guards* P_2 and P_3 with local states $p_2.v$ and $p_3.v$ respectively.

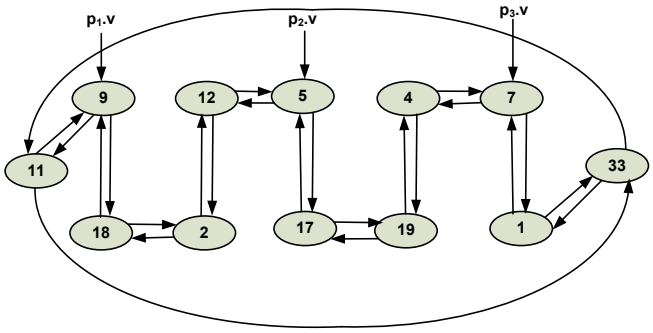

Figure 3: The doubly circular linked-list configurations of P_1, P_2 and P_3 initialised with elements from set S. Each copy of the list is also initialised with an initial pointer location ($p_1.v$, $p_2.v$, or $p_3.v$) respective to the process it is sent.

3.2 Adapting a Knapsack problem instance for distributed computing obfuscation

We now consider an instance of a hard combinatorial problem called Knapsack problem (Garey & Johnson 1979) and show that it can be adapted for manufacturing distributed opaque predicates. The original 0/1-Knapsack problem can be stated as follows:

Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of positive integers and a sum $T = \sum_{i=1}^{n} x_i a_i$ where each $x_i \in \{0, 1\}$, find x_i. This decision problem has been shown to be NP-complete. In adapting this problem for manufacturing distributed opaque predicate, the obfuscator selects the set S of positive integers and x_i's according to some predetermined sum T. An adversary through careful static analysis and reverse engineering may come to learn about set S and sum T. However, given an arbitrarily large set, the hard problem for him is to not only decide if a solution vector x exists but to also to determine the vector at precisely the program points $(t_1, t_2, ...)$ where distributed opaque predicates are used to obscure the control-flow of P_1. This is hard since the distributed opaque predicates are constructed from local states (the values range in set S) of *guard* processes and P_1 and local states dynamically change depending on the interaction pattern between processes. This underlying concept will gradually evolve as we describe our methodology.

For our illustration, we select an arbitrary set as:

$$S = \{11, 9, 18, 2, 12, 5, 17, 19, 4, 7, 1, 33\}$$

After dynamically selecting/spawning the *guards*, process P_1 and the *guards* are each initialised by passing a dynamic data structure, such as a doubly circular linked-list, initialised with the elements of the set S. This is illustrated in Figure 3.

In addition to initialising the linked lists with elements of set S, each copy is also initialised with an initial pointer location respective to the process the

list is sent. Node values corresponding to the pointer locations form the local state of that particular process. For our illustration with three processes, the list is initialised with three pointers: $p_1.v$ for P_1, $p_2.v$ for P_2, and $p_3.v$ for process P_3. Messages are exchanged between the *guards* $\{P_2, P_3\}$ and P_1 according to an embedded communication pattern. Generation of this predetermined communication pattern will be discussed shortly. Considering an arbitrary sum for our illustration as $T = 27$, the corresponding solution vector x for the sum T is:

$$x = \{0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0\}$$

For our three process illustration, this solution vector corresponds to $p_1.v = 18$, $p_2.v = 5$, and $p_3.v = 4$. The *distributed opaque predicate* (Φ) thus formed in this case would be:

$$\Phi : p_1.v + p_2.v + p_2.v = 27$$

In the following subsections, we will explain how to coordinate the local state update values between processes by controlling their interaction pattern such that Φ is satisfied at precisely the program points where the obfuscator decides. We note that there could be other possible solution vectors for set S and a similar approach for constructing distributed opaque predicates could be used with different sets of *guards* and the same sum T.

3.3 Defining the local state update rules

The local state update rules defined on inter-process message communication events are defined as follows:

`receive(m)` $\Longrightarrow$ `pointer shifted right of the current node`

`send(m)` $\Longrightarrow$ `pointer shifted left of the current node`

Thus, if the local state of P_1, defined by $p_1.v$, is 9 at a certain point in P_1's execution and if P_1 receives a message, the local state will change to 18. Similarly, if P_1 sends a message, the local state changes to 11. Since the distributed opaque predicate is constructed by composing the local states of individual processes and each local state value (corresponding to the pointer location) fluctuates when processes send or receive messages, the predicate will alternate between true/false outcomes throughout the run.

3.4 Selection of communication pattern and message types

As stated earlier, local state update of individual processes take place according to an embedded invariant communication pattern. This predetermined pattern is generated when the embedded *send/receive* calls in processes P_1, P_2, and P_3 get executed. The calls could be embedded by the obfuscator by tracing and annotating the processes with *send/receive* primitives, much in the same way dynamic watermarking algorithms annotate programs for inserting watermark building code (Nagra & Thomborson 2004). However, there are a couple of problems with adopting this approach for embedding the communication pattern. First of all, embedded *send/receive* calls will generate an arbitrary pattern for each run of the program unless they are controlled in some way. Secondly, because of the nondeterminism and latency associated with asynchronous message passing, there is no guarantee of causal delivery of messages. Thus, we have to ensure message exchanges satisfy FIFO (First-in-first-out) delivery. This delivery order ensures for all messages m and m':

$$\text{send}_\text{i}(\text{m}) \rightarrow \text{send}_\text{i}(\text{m}') \Rightarrow \text{deliver}_\text{j}(\text{m}) \rightarrow \text{deliver}_\text{j}(\text{m}')$$

In distributed systems, the notion of global clock is absent. We propose using vector clocks (Mattern 1989) for solving these two problems. Using vector clock, event orderings based on increasing clock values are guaranteed to be consistent with causal precedence. Before going into a detailed discussion on its usage for constructing the predetermined communication pattern, we briefly provide a general overview of vector clocks.

Vector clock of a system of n processes is an array of n logical clocks, one per process. A local copy of the vector clock is kept in each process P_i, contributing a local state in the construction of distributed opaque predicate. A notation of $VC_i^b[i]$ denotes the logical clock value of P_i at *send/receive* event b. $VC_i^b[j]$ denotes the time $VC_j^a[j]$ of last event a at P_j that is known to have happened before its local event b. The vector clock algorithm update rules could be specified as:

- If a and b are successive events in P_i, then $VC_i^b[i] = VC_i^a[i] + 1$.
- Also, if b denotes *receive(m)* by P_i with a vector timestamp t_m, then $VC_i^b[k] = max\{VC_i^a[k], t_m[k]\}$, for all $k \neq i$.

Three obfuscation-specific message classes are used for message exchanges between process P_1 and the *guards* P_2 and P_3. These message classes help in maintaining consistency of vector clock values and local state updates. Each class is identified by a special tag. The first class is identified by the tag `SYSTEM`. Messages of this class may originate in either P_1, P_2 or P_3 and carry vector timestamp in them. Also, when a process participates in a *send/receive* event of `SYSTEM` type messages, it updates its vector clock and local state according to the state update rules specified in the previous subsection. The second class, `REQUEST`, type message may only originate at P_1 since it is used to request local state values for the *guards*. This class also carries vector timestamp and causes vector clock updates but does not cause any change in local state when received by the *guards*. The third type of message is identified by tag `RESPONSE`. This type is identical to the second class of messages with the exception that these originate at *guards* and are received by P_1. `RESPONSE` messages are used by *guards* to send local state values back to P_1 against incoming `REQUEST` messages.

An example of predetermined communication pattern is illustrated with processes P_1, P_2, and P_3 in the event-time diagram of Figure 4. An event-time diagram maps each event against time and state changes are effected by exchange of messages. The embedded *send/receive* calls in processes P_1, P_2, and P_3 generate this communication pattern. The vector clock value for each participating process is indicated within the square brackets and the value of local state is indicated in variable $p_i.v$, where i denotes the process number. Thick arrows in the figure denote `SYSTEM` type messages. Thin arrows denote `REQUEST` type messages and dashed arrows represent `RESPONSE` type messages.

As evident from Figure 4, asynchrony of message passing induces concurrency within the system. Because of this concurrency, an adversary will find it difficult to monitor local state changes occurring between the processes from outside and determine if distributed opaque predicate (Φ) is satisfied at a particular program location in a particular run. Along the

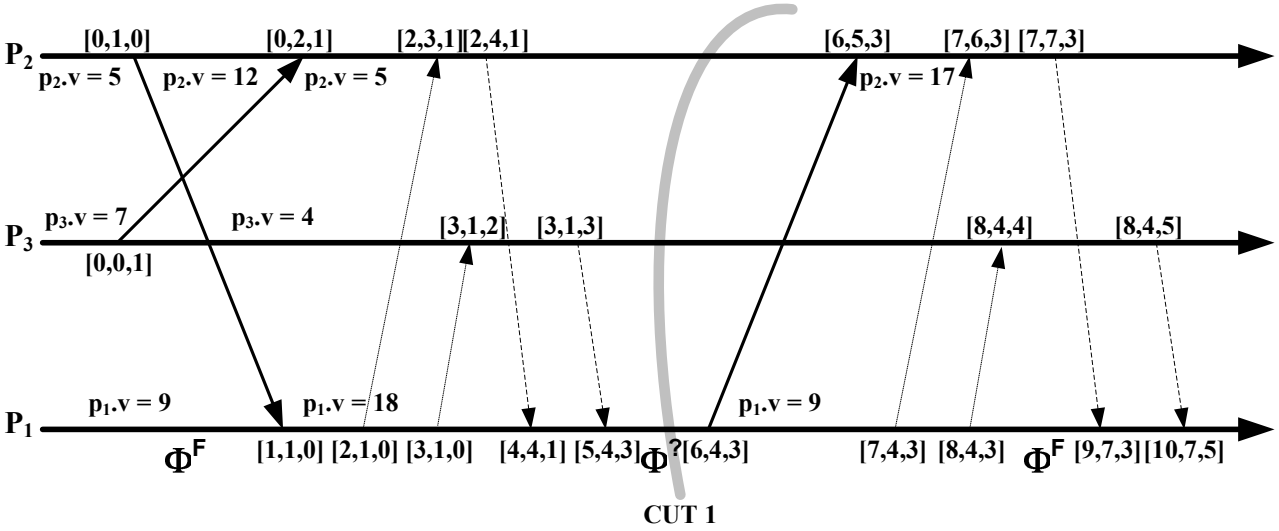

Figure 4: The invariant in the form of a predetermined nondeterministic communication pattern is embedded by the obfuscator into P_1, P_2 and P_3. The update pattern of local states can be traced from Figure 3. Thick arrows denote `SYSTEM` type messages, thin arrows denote `REQUEST` type messages, and dashed arrows denote `RESPONSE` type messages.

timeline of process P_1, we have labeled the value of predicate (Φ) between two successive events distinguished by vector clock values. A (Φ^T) label implies that the predicate is guaranteed to hold true within that event interval (successive events) for that particular run. Similarly, (Φ^F) implies that the predicate is guaranteed to be false within that interval for that particular run. A label denoted by ($\Phi^?$) along the timeline implies that the predicate value is unknown since Φ is not guaranteed to hold.

Moreover, in Figure 4, it seems that (Φ) would be satisfied at `CUT1` since the local states $p_1.v = 18$, $p_2.v = 5$, $p_3.v = 4$ add up to 27. However, note that there is no guarantee of Φ being satisfied at `CUT1`. This guarantee cannot be made because of the following two special cases that could arise out of nondeterminism:

3.4.1 No guarantee on message delivery order

Consider the case from Figure 4 where the message (henceforth referred to as message a) originating from P_3 at vector clock value $[0, 0, 1]$ reaches before the message (henceforth referred to as message b) originating at vector clock value $[0, 1, 0]$ of P_2 is sent by process P_2. This situation is depicted in Figure 5.

When message a reaches *guard* process P_2, the vector clock value changes to $[0, 1, 1]$ and P_2's local state, $p_2.v$, changes from 5 to 17 (refer to Figure 3). *Guard* process P_3's local state, $p_3.v$, changes from 7 to 5. However, after P_2 sends message b at vector clock $[0, 2, 1]$, its local state reverts to 5. When the probe messages (`REQUEST`) are sent by P_1 after the vector clock state $[1, 2, 1]$, the local state values returned from processes P_2 and P_3 are $p_2.v = 5$ and $p_3.v = 4$ respectively. By the time the probe messages are sent to P_2 and P_3, process P_1 has already changed its local state value, $p_1.v$, to 18. The local state values of processes P_1, P_2, and P_3 add up to 27 and the distributed opaque predicate (Φ) is satisfied at `CUT1`.

3.4.2 No guarantee on message delivery

Now consider the case where after the first receive of message b at $[1, 1, 0]$ by process P_1, it cannot be

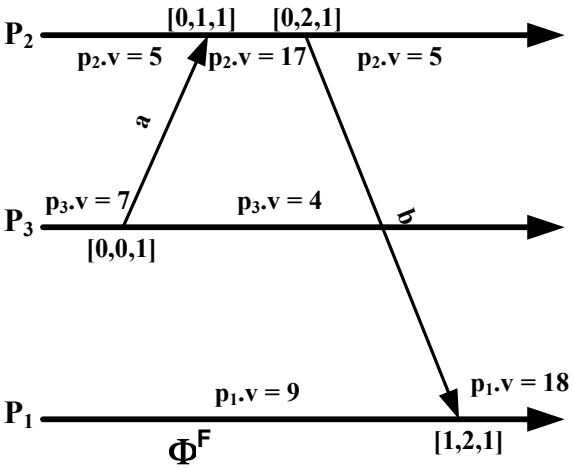

Figure 5: No guarantee on message delivery order. Messages a and b are swapped and (Φ) is satisfied at `CUT1`.

guaranteed that the message a from *guard* process P_3 originating at $[0, 0, 1]$ has reached *guard* process P_2. This guarantee cannot be made because of the nondeterministic nature of asynchronous message-passing. This situation is depicted in Figure 6.

As seen from the figure, since *guard* process P_2 changes its local state to $p_2.v = 12$, the local state values of processes P_1, P_2, and P_3 do not add up to 27. Consequently, the distributed opaque predicate (Φ) is not satisfied at `CUT1`. In yet another specialization of this case, message b may reach process P_1 even before message a originates from *guard* process P_3. In this case, *guard* process P_3 will maintain its local state value at $p_3.v = 7$. Thus, the predicate value will also not be satisfied in this case since the sum of the local state values of processes does not add up to 27.

Thus, we can generally observe, from the nondeterministic communication pattern of Figure 4, that while designing the communication invariant, crossover message-passing patterns will cause nondeterminism within the system and this property could be utilized by the obfuscator to confuse attackers into falsely believing that a distributed opaque predicate will be guaranteed to hold true or false at a particular

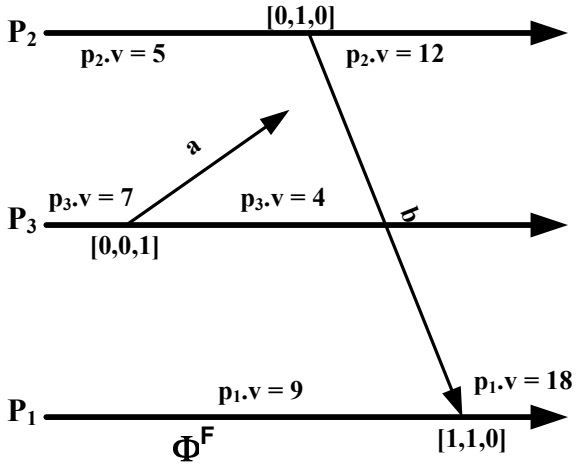

Figure 6: No guarantee on message delivery. Message a is in transit while message b reaches process P_1. The predicate (Φ) is not satisfied at `CUT1`.

program location.

On the other hand, deterministic communication patterns would produce guaranteed results for distributed opaque predicates. An example of deterministic cyclic communication pattern is shown in Figure 7. This event-time diagram is a continuation of the one shown in Figure 4 and the vector clock ticks are continued along the timelines of processes P_1, P_2, and P_3. At $[14, 10, 9]$, process P_1 is ready to evaluate the distributed opaque predicate (Φ). Interestingly at this point, it can be guaranteed that there are no messages in transit and hence the predicate must hold true (Φ^T) at `CUT2`. For all other event intervals, (Φ) is guaranteed to be false (Φ^F).

3.5 Distributed opaque predicate embedding and guarded commands for maintaining local state consistency

Just as nondeterminism and asynchrony can be used as tools against the adversary, these could also cause problems to the obfuscator since uncontrolled concurrency will update states in an unpredictable way. If local states of processes are updated in an uncontrolled way, then distributed opaque predicates cannot be used effectively for control-flow obfuscation.

The problem associated with unpredictable local state update can be brought under control if the communication pattern generating code (specifically the *send/receive* primitives) can be guarded; i.e., a message contributing to a deterministic communication pattern is only sent from a process if it is guaranteed that the vector clock value of the process issuing this *send* is up-to-date. Alternatively, this means that the process should have completed all the message communication events (*send/receive*) before issuing another *send*. We show an abstract pseudo-code for controlled message passing and predicate evaluation for process P_1 in Figure 8. We have used blocking *receive* to ensure that the local state of process P_1 is consistent before it issues a *send* message (i.e., the process busy-waits on all outstanding messages it has not yet received). To make it more flexible, non-blocking *receive* with guarded *sends* could be used to maintain consistency of local states. This can be implemented by making sure that before each *send* primitive, the vector clock from last *receive* is up-to-date (by comparing it against an expected timestamp value). If the clock is not up-to-date, the process blocks the *send* call for outstanding *receives*.

Figure 8 shows pseudo-code snippets for nondeterministic (`CUT1`) and deterministic (`CUT2`) evaluation of the predicate Φ. At `CUT1`, Φ is unknown and hence dummy actions are inserted in branches corresponding to both '`true`' and '`false`' paths. However, at `CUT2`, the obfuscator knows that Φ holds true (because it knows when it participates in deterministic and nondeterministic message-passing) and hence inserts real actions in the path corresponding to the '`true`' branch of the control statement. Pseudo process interaction codes for *guard* processes are similar to P_1's code and have been excluded from this contribution because of space limitations.

During obfuscation phase of P_1, the obfuscator may embed many distributed opaque predicates at different control-flow points in the program corresponding to, for example, the construction of watermarking code. Any arbitrary nesting of distributed opaque predicates can be used for obfuscating the control-flows. A different set of *guard* processes could also participate in different communication invariants involving other local state update rules.

4 Security analysis of obfuscation using distributed opaque predicates

In this section, we comment on the obfuscatory strength of the proposed technique by arguing that known forms of static analysis attacks and a restricted class of dynamic analysis attack are intractable from an adversary's perspective. For each class of attack, we also present our assumptions on technical limitations of the adversary.

4.1 Static analysis attacks

We argued in section 2 that static analysis of temporally unstable distributed opaque predicates will not reveal their outcome since the invariant communication pattern which influences their outcome is generated from the embedded *send/receive* primitives on-the-fly. We did not, however, comment on the difficult issues an attacker needs to address in order to statically *find* these distributed opaque predicates from the process codes.

In order to statically analyse the obfuscated code, an adversary must depend on static slicers to slice parts of the process code which could affect the value of distributed opaque predicates at obfuscated control-flow points. 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 (Horowitz 1997, Hind *et al.* 1999) to determine the kind of structure the local state of processes points-to (this information he could get from the message parameters), and if the pointer corresponding to the variables used in the construction of distributed opaque predicates refer to the same dynamic data structure in the *guards* at some program location (where the distributed opaque predicates are used in P_1). To achieve this, static analysers must use inter-process escape alias analysis to determine the objects that can be referenced in processes separate from the ones in which they are allocated.

Though much research work on intra- as well as inter-procedural alias analysis and inter-procedural thread escape analysis have been done in the last few years (Rugina & Rinard 1999, Sălcianu & Rinard 2001, Whaley & Rinard 1999), we have been unsuccessful in finding a technique 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

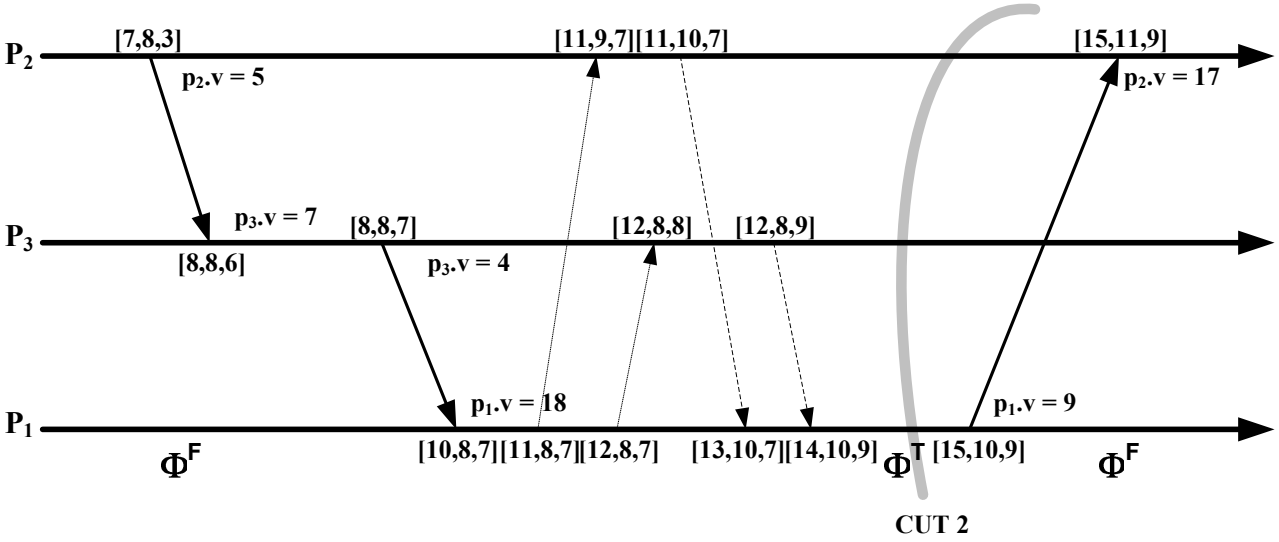

Figure 7: The invariant in the form of a predetermined deterministic communication pattern is shown in this figure. As before, thick arrows denote `SYSTEM` type messages, thin arrows denote `REQUEST` type messages, and dashed arrows denote `RESPONSE` type messages.

addressed by the program analysis community is because we do not have efficient, precise and scalable algorithms for performing simpler cases of alias analysis in sequential multi-threaded programs and asynchronous concurrent systems present problems that are much greater in magnitude.

4.2 Dynamic analysis attacks

Dynamic analysis attacks assume that the adversary has most (if not all) of the static analysis information available since he has to monitor the local state value changes of individual processes (an assumption we argued in the previous subsection as quite intractable). Moreover, in order to mount a dynamic analysis attack, the adversary needs to learn about the structure of the distributed opaque predicate so that he can identity which processes are contributing in building the global state (i.e. the *guard* processes along with process P_1). We need to make the restriction that an adversary cannot possess sufficient static analysis information in order to insert debugging probes at obfuscated control-flow points of P_1. If an adversary is able to do this, he can quite easily determine the outcome of distributed opaque predicates by just checking probe values during execution of process P_1. If we fail to make this restriction, the use of opaque predicates in any form of program obfuscation would be trivial. We, however, make the relaxation that the adversary can monitor the communication events using *sniffer* processes in order to monitor individual local states formed by process P_1 and the *guards*. This scenario is depicted in Figure 9.

We now show proceed to show that the problem of global state monitoring is hard even if the adversary manages to collect all necessary static analysis information. The problem of distributed opaque predicate evaluation, from the adversary's perspective, can be stated as evaluating predicate Φ as a function of the global state of a distributed system. It is problematic to detect unstable distributed opaque predicates since the condition encoded by the predicate may not persist long enough for it to be true when the predicate is evaluated by the adversary. The domain of such predicates is a Boolean valued function formed on the set of all possible cuts from all possible executions of the distributed system. Therefore, the predicate detection problem can also be defined as identifying a cut

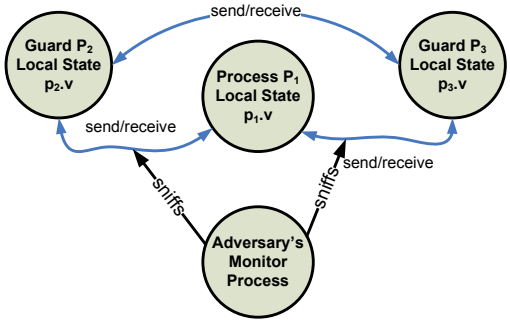

Figure 9: A typical dynamic analysis attack scenario by actively monitoring state changes to detect distributed opaque predicate Φ.

in which the predicate evaluates to true. The difficulty associated with detection is the fact that the number of states from any execution may be exponential in the number of processes.

Let $\{X_1, X_2, \ldots, X_n\}$ define a sequence of cuts, where for all i, $X_i < X_{i+1}$. A sequence of cuts is called an *observation* if and only if for all i, X_i and X_{i+1} differ by exactly one state. The adversary has to detect a consistent observation O of the distributed computation such that Φ holds in a global state of O. Now, this is a decision problem in the form of:

`Given:` an execution Y of n processes, an initial cut $X \leq Y$, and the predicate Φ.

`Determine:` if there exists a cut $W : X \leq W \leq Y$ such that $\Phi(W)$ is true.

Chase & Garg (1995) proved that this detection problem is NP-complete by showing that the detection of general global predicate is intractable even for simple distributed computation where the local states are restricted to take only `true` or `false` values and no messages are exchanged within the system.

In the subsections to follow, we model a dynamic analysis distributed monitoring attack by discussing in details the three types of available algorithms an adversary may choose to dynamically evaluate the outcome of Φ and the technical limitations these algorithms possess.

Process P_1:

```
initialize(VectorClock); //Initialize Vector Clock to [0,0,0]

… //Start nondeterministic predicate evaluation

//get SYSTEM message
while(!receive(VectorClockTimeStamp,SYSTEM,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [1,1,0]
shift_right(p1.v); //point to the right node

//probe for local states from guard processes
increment(VectorClock); // Vector Clock value [2,1,0]
send(P2,REQUEST); //probe for p2.v value
increment(VectorClock); // Vector Clock value [3,1,0]
send(P3,REQUEST); //probe for p3.v value

//get RESPONSE messages
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p2.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [4,4,1]
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p3.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [5,4,3]

// evaluate distributed opaque predicate
if (p1.v+p2.v+p3.v==27) {// CUT1: Predicate Value Unknown
    // Dummy watermark building code
}
else {
    // Dummy watermark building code
}

… //Start deterministic predicate evaluation

//get SYSTEM message
while(!receive(VectorClockTimeStamp,SYSTEM,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [10,7,8]
shift_right(p1.v); //point to the right node

//probe for local states from guard processes
increment(VectorClock); // Vector Clock value [11,8,7]
send(P2,REQUEST); //probe for p2.v value
increment(VectorClock); // Vector Clock value [12,8,7]
send(P3,REQUEST); //probe for p3.v value

//get RESPONSE messages
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p2.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [13,10,7]
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p3.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [14,10,9]

// evaluate distributed opaque predicate
if (p1.v+p2.v+p3.v==27) {   // CUT2: Predicate Value True
    // Real watermark building code
}
else {
    // Dummy watermark building code
}
```

Figure 8: Pseudo-code showing the obfuscation of control-flow in P_1 using distributed opaque predicate Φ.

4.2.1 Active monitoring by taking snapshot

The first option the adversary has is to solve the global predicate evaluation problem through *active* monitoring. In this strategy, the adversary uses a monitor process which sniffs the communication between process P_1 and the *guards* at some predetermined periodic intervals and then combines all the local states obtained to build the global state. This strategy is called *'snapshot'* approach and Chandy & Lamport (1985) describe an algorithm to construct consistent global states using snapshots of individual local states. Since communication within distributed systems incurs latency, the consistent global states thus constructed can only reflect some past state of the system. By the time the snapshots are obtained, conclusions drawn about the system by evaluating the distributed opaque predicate may have no bearing to the present. Therefore, the snapshot algorithm is suitable for monitoring predicates that do not change value throughout the entire program run and since our method uses temporally unstable predicates, the adversary will not be able to deduce a correct reasoning about the predicate's behaviour using this algorithm - the predicate may have held even if it is not detected.

4.2.2 Passive monitoring by constructing state lattice

Through the second approach, due to Cooper & Marzullo (1991), the adversary can collect all local state values from individual processes and check for consistent observation O using *passive* monitoring. In order to implement this algorithm, the adversary's monitoring process must sniff the *guard* processes and P_1 for portions of their local states that are referenced in Φ. The monitor maintains sequences of these local states, one sequence per process, and uses them to construct the global state. This procedure is based on incrementally constructing the lattice of consistent global states associated with the distributed computation. The state lattice formed is linear in the number of global states, and the number of global states formed is $O(e^n)$ where e is the maximum number of events monitored and n is the number of processes in the system. For every global state in the lattice, there exists at least one run that passes through it. Hence, if any global state in the lattice satisfies Φ, the distributed opaque predicate is detected.

The problem with this type of monitoring is that the adversary may end up incorrectly including spurious local state changes in case he erroneously considers processes that are interacting with *guards* and P_1 during construction of the lattice or if he includes spurious message communication events (by failing to distinguish between message classes that only update the vector clock values and not the local state of processes). Hence, if the number of *guards* in the system is large and a considerable amount of message exchange takes place, the adversary will face the problem of state explosion while trying find a consistent cut by 'walking-through' the lattice thus formed. Moreover, if the adversary fails to monitor some of the process interactions, the amount of concurrency in the form of local state changes will increase and this will, in turn, increase the states of the lattice. Increase in the number of *guards* in the system will increase the dimension of the lattice proportionately. Furthermore, the adversary has to repeat this passive monitoring process to detect the outcome of each distributed opaque predicate used to obfuscate control-flow points in process P_1. The complexity will further increase if such predicates are nested and *guards* are spawned dynamically during P_1's execution.

4.2.3 Active monitoring by exploiting predicate structure

In the final approach, the adversary can use Garg & Waldecker's (1994) method for detecting unstable global predicates. Their method exploits the structure of predicate by decomposing the predicate into a conjunction of local predicates and independently detecting the outcomes of these local predicates. It also requires the use of explicit token passing messages between the monitor process and the processes which contribute states in the construction of distributed opaque predicates. This approach works well for predicates that are conjunctive in nature. However, for relational distributed opaque predicates, their method yields no feasible solution because relational predicates cannot be broken down into conjunction of predicates formed on local states. Moreover, it requires processes participating in maintaining the distributed opaque predicate to cooperate with adversary's monitor process by maintaining snapshots (evaluating their component of the predicate) and passing the result and dependence information to the adversary's monitoring process. This requirement quite unrealistic under reasonable practical assumptions.

We conclude by observing that out of these three available approaches, the adversary has to resort to using only the second approach because the other two available approaches are only suitable for detecting either predicates that do not change their value during the entire program run or predicates that can be broken down into a conjunction of local predicates. Moreover, the second approach will be intractable if a large number of *guard* processes are used or are spawned dynamically during execution of the obfuscated process P_1. Also, in the absence of precise static analysis methodologies, spurious events and state changes would be erroneously taken into consideration by the adversary and this would make the detection process incorrect and intractable. Under pragmatic assumptions, we believe practical distributed systems will employ a large number of processes as *guards* and hence processes obfuscated with distributed opaque predicates will be resilient to passive monitoring dynamic analysis attacks.

5 Conclusion

In this contribution, first of its kind, we have addressed the problem of code obfuscation in software executing in distributed computing environments. Specifically, we have addressed control-flow obfuscation and have extended the original concept of opaque predicates proposed by Collberg *et al.* (1998) to the domain of distributed computing. We have demonstrated that hard combinatorial problems can be tuned with open problems related to distributed systems state monitoring to manufacture a new class of resilient opaque predicates; which we defined in this contribution as distributed opaque predicates. We have also demonstrated through a detailed security analysis that our class of distributed opaque predicates is resilient to known static analysis attacks and passive monitoring dynamic analysis attack from adversaries.

The following salient points could be noted regarding this new class of opaque predicates:

- `Stealth:` The relational structure of distributed opaque predicates will make these unobvious to an attacker since predicates of this nature appear as conditional expressions in most programs. Moreover, *guard* processes, along with

the process to be obfuscated, maintain the distributed opaque predicate invariant through an embedded communication pattern. This pattern is generated by *send/receive* calls embedded within process code. Processes in loosely-coupled distributed systems, as such, communicate using message-passing and hence the presence of additional *guards* and their interactions with the host process will, we believe, be unsuspecting from the perspective of an adversary.

- `Performance:` Distributed systems are inherently loosely-coupled in nature and do not enforce hard timing requirements on task completion. Hence, the overall slowdown in system effectuated by additional *guard* processes and message exchanges might be acceptable to developers having stringent security requirements.

As part of our future work, we will concentrate on automatic embedding of distributed opaque predicates at selected control-flow locations in distributed computing processes through program annotation. We would also come up with a model of making the obfuscated system, consisting of the obfuscated process and cooperating *guards*, more fault-tolerant such that the system can function in case one or more *guards* are accidentally lost or purposefully killed by an adversary. Our present model is rigid in the sense that the loss of *guard* processes will make the obfuscated program go into an incorrect state, thus adding some form of tamper-proofing. But, this notion is weak since the *guards* and messages may be lost in the system accidentally. We will also investigate into new classes of distributed opaque predicates and instances of hard combinatorial problems for generating them in the future.

References

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., & Yang, K. (2001), On the (Im)possibility of Obfuscating Programs, *In* the proceedings of CRYPTO-2001. LNCS Volume 2139, Springer-Verlag. Santa Barbara, CA, USA

Chow, S., Gu, Y., Johnson, H. & Zakharov, V.A. (2001), An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. *In* the proceedings of 4^{th} International Conference on Information Security, LNCS Volume 2200. Springer-Verlag. Malaga, Spain.

Garey, M. R. & Johnson, D. S. (1979), A guide to the theory of NP-completeness. W.H. Freeman and Company.

Thomborson, C., Nagra, J., Somaraju, R. & He, C. (2004), Tamper-proofing software watermarks. *In* the proceedings of 2^{nd} workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalization. Volume 32. Dunedin, New Zealand. ACM Digital Library.

SETI@home (2005), http://setiathome.ssl.berkeley.edu/ (accessed July 15 2005).

Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q. & Zhang, Y. (2000), Experience with software watermarking. *In* the proceedings of 16^{th} IEEE Annual Computer Security Applications Conference (ACSAC'00). IEEE Press. New Orleans, LA, USA.

Nagra, J. & Thomborson, C. (2004), Threading Software Watermarks. *In* the proceedings of 6^{th} International Workshop on Information Hiding, LNCS Volume 3200, Springer-Verlag. Toronto, ON, Canada.

Hohl, F. (1998), Time limited blackbox security: Protecting mobile agents from malicious hosts. *In* the proceedings of 2^{nd} International Workshop on Mobile Agents, LNCS Volume 1419, Springer-Verlag. Stuttgart, Germany.

Collberg, C., Thomborson, C. & Low, D. (1998), Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *In* the proceedings of 1998 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). San Diego, CA, USA.

Lamport, L. (1978), Time, clocks and the ordering of events in a distributed system. *In* Communications of the ACM, 21(7):558-565.

Chase, C. & Garg, V.K. (1995), Detection of global predicates: Techniques and their limitations. *In* the Journal of Distributed Computing, Volume 11, Issue 4, pages 191 - 201. Springer-Verlag.

Mattern, F. (1989), Virtual time and global states of distributed systems. *In* the proceedings of Workshop on Parallel and Distributed Algorithms, Elsevier Science Publication, pages 215-226.

Horowitz, S. (1997), Precise Flow-insensitive may-alias in NP-hard. *In* ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 19 No. 1.

Hind, M., Burke, M., Carini, P. & Choi, J.D. (1999), *In* ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 21 No. 4.

Rugina, R. & Rinard, M. (1999), Pointer analysis for multithreaded programs. *In* the proceedings of 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99). Atlanta, GA, USA.

Sălcianu, A. & Rinard, M. (2001), Pointer and escape analysis for multithreaded programs. *In* the proceedings of 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '01), Snowbird, UT, USA.

Whaley, J. & Rinard, M. (1999), Compositional pointer and escape analysis for Java programs. *In* the proceedings of 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, CO, USA.

Chandy, K.M. & Lamport, L. (1985), Distributed Snapshots: Determining global states of distributed systems. *In* ACM Transactions on Computer Systems, pages 63-75.

Cooper, R. & Marzullo, K. (1991), Consistent detection of global predicates. *In* the proceedings of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91). Santa Cruz, CA, USA.

Garg, V. K. & Waldecker, B. (1994), Detection of weak unstable predicates in distributed programs. *In* IEEE Transactions on Parallel and Distributed Systems, pages 299-307, Volume 5, No. 3.