

Applying Static Analysis to Verifying Security Properties

Xiaolan Zhang and Trent Jaeger and Larry Koved
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
{cxzhang,jaegert,koved}@us.ibm.com

Abstract

While developing “bug-free” software is impossible in general, with recent advances in static analysis techniques it is feasible to verify certain classes of security properties for large software systems, thus greatly enhancing end users’ confidence on the software. In this paper we identify an important class of security properties that can be reduced to the classical *dominance problem* and show how they can be checked efficiently and accurately, using a dominance algorithm implemented on top of the JaBA analysis framework.

1 Introduction

As software systems become increasingly complex, the security issues accompanying them become increasingly aggravated. It is generally understood that developing a “bug-free” application of reasonable size is no longer possible. It is however, still possible to verify a certain class of security properties for large software systems, such that the verification can provide some level of assurance to the user of the software. This is similar to the *Common Criteria Certification Evaluation and Validation Scheme* [10], where one goes through a certification process that verifies that the target software satisfies certain level of security requirements.

Traditionally, people have used model checking for verifying security properties [2, 3, 8]. Unfortunately, model checking is a time consuming process. In addition, model checking does not work on the implementation directly, rather, the model is extracted from the implementation and represents the abstract states of the program. Thus, errors might be introduced during the mapping from the actual implementation to the abstract model. In summary, it is not practical to model check very large software systems, unless the security properties being checked are sufficiently simple.

In the past few years some progress has been made in applying static analysis techniques to software

verification [5, 6, 11]. These systems are more scalable than traditional model checkers. However, they are quite limited in their checking ability. For example, the tools built by Jensen et. al. [11] can check for security properties that involve only control flow. Our goal, on the other hand, is to build an analysis framework that is both scalable and capable of checking a wide range of security properties, including properties that involve both control and data flow.

Tools exist that are capable of identifying security and other flaws in system software [9, 7]. But their main purposes are to discover flaws, rather than verifying security properties (prove the lack of flaws). Identifying security flaws is easier than verifying security properties: identification aims at finding some flaws while verification must find all flaws. Verifying a security property requires the development of a model that encompasses all relevant concepts and an analysis that is *complete* (i.e., has no false negatives) and generates a manageable number of false positives. Thus, verification requires more accuracy both in model development and analysis.

Since security-critical software is often large, the security analysis tool must be able to scale to thousands of lines of code, at the same time remaining precise enough so it can be used for verification purposes. Thus a good analysis tool must make the right trade-off between accuracy and performance.

In this paper we identify an important class of security properties that can be reduced to the classical *dominance relationship* checking problem. We argue that static analysis tools can be built that are both scalable and accurate for the purpose of verifying complex security properties (e.g. dominance relationship) of large realistic applications. We demonstrate our claim by presenting a general algorithm that verifies dominance relationships, and implementing the algorithm on the JaBA static analysis framework. Our experimental evidence shows that it produces significantly fewer false positives compared with a previous approach based on CQUAL [15].

The rest of the paper is organized as follows. Section 2 describes the problem of verifying dominance relationship. Section 3 presents our solution. Section 4 shows the experimental results and Section 5 concludes the paper.

2 Dominance Relationship

We say a program point v *dominate* w if every path from the beginning of the program to w includes v [13]. In addition, it is said that a program point y *postdominate* program point x if every path from x to the end of the program includes y . Simple as it looks, a large set of security properties can be reduced to the problem of checking whether a dominance relationship holds between two events.

The simplest class of security properties that can be reduced to a dominance problem are those that involve only control flows. For example, the proper way to create a *chroot jail* requires calling `chdir("/")` immediately after calling `chroot()` [5].

Security properties that involve data flows are more common. A classical example is checking for unmatched spin locks in the Linux kernel which might cause deadlocks [8]. The Linux kernel uses two spin lock functions, `spin_lock(spinlock_t *lock)` and `spin_unlock(spinlock_t *lock)`, respectively, for locking and unlocking. Checking for unmatched lock/unlock pairs can be translated into the problem of checking for a dominance relationship between `spin_lock` and `spin_unlock` and a postdominance relationship between `spin_unlock` and `spin_lock`. The difference between this class of properties and those that only involve control-flow is that in this case one needs to track the variable `lock` in addition to checking the dominance relationship, hence the name data flow.

In this paper, we focus on a class of security properties that are reducible to the dominance problem but that require a more flexible definition of the events to be checked. Thus checking these properties require a more complex analysis engine. Examples include:

- Verification of complete mediation of a reference monitor interface.
- Verification of complete coverage of an auditing facility.
- Verification of correct memory object reuse.

These example security properties are significant because they can provide users with the assurance that the specified security properties are always met.

```

1:  /* Code from fs/read_write.c */
2:  sys_lseek(unsigned int fd, ...) {
3:      struct file * file = fget(fd);
4:      ...
5:      retval = security_ops->file_ops->llseek(file);
6:      if (retval) {
7:          // failed check, exit
8:          goto bad;
9:      }
10:     // passed check, perform operation
11:     retval = llseek(file, ...);
12:     ...
13: }

```

Figure 1: An example of LSM hook.

An example of reference monitor interfaces is the Linux Security Modules (LSM) interface [14]. LSM defines an interface for flexible, mandatory access control in the Linux kernel. LSM consists of a set of generic authorization hooks that are inserted into the kernel source that enable kernel modules to enforce system access control policy for the kernel. Thus, the Linux kernel is not hard-coded with a single access control policy. Module writers can define different access control policies, and the community can choose the policies that are most effective for their goals.

The code segment in Figure 1 shows an example of how LSM hooks are inserted in the kernel. The function `sys_lseek()` implements the system call `lseek`. The security hook, `security_ops->file_ops->llseek(file)` (line 5), is inserted before the actual work (the call `llseek()` at line 11) takes place. The goal is to check that all security-sensitive operations (e.g., `llseek()`) are dominated by a check to the reference monitor (e.g., the security hook at line 5).

Similarly, in the case of auditing, we would like to verify that all security-sensitive operations are dominated by an audit call. In the case of object reuse, we would like to verify that the release of the memory occupied by a security-sensitive object is preceded by steps that properly remove all sensitive information.

3 Verifying Dominance Relationships

3.1 Computing Dominance Relationships

The algorithm for computing dominators intra-procedurally is well known and has been documented in many compiler textbooks [1, 13]. Our algorithm for computing dominance relationships makes several enhancements over the classical dominator algorithm. First, our algorithm is inter-

procedural. Secondly, our algorithm takes into consideration data objects along with the events. Specifically, we leverage data flow analysis to verify that the data objects in the dominator event and those in the dominatee event form a relationship as specified in the security property.

To better understand the algorithm, we use a simple example to illustrate the main points of the algorithm. Figure 2(a) shows a hypothetical program containing five functions: `main()`, `f1()`, `f2()`, `CHECK()`, and `DOP()`. The function `CHECK()` performs the authorization check; and the function `DOP()` (stands for dangerous operations) contains potentially dangerous manipulations of critical data structures, and thus needs to be mediated. More specifically, all invocations of function `DOP()` must be preceded by a call to function `CHECK()`. Thus this simple code example represents the class of complete mediation problem (see Section 2. The dominator here is the `CHECK` function, and the dominatee is the `DOP` function. Figure 2(b) shows the corresponding *super graph*, defined as a call graph overlaid with intra-procedural control flow graph. Each node of the inter-procedural super-graph is a function (ignoring the labels on the edges for now). Intra-procedurally, each node can be represented by a control flow graph of basic blocks.

Our dominator algorithm consists of three stages. In the first stage, we collect information about the dominators within each node (intra-procedural analysis). We then propagate the dominators up the call graph (intra- and inter-procedural analysis) in the second stage. In the final stage, we compute for each dominatee the set of dominators (intra- and inter-procedural analysis). We next describe these three stages in more detail.

Stage 1. Local Dominator Set Generation.

In this stage, we identify local dominators within each node in the super graph. Since this computation is performed entirely within each node, this process is an intra-procedural analysis. We use $LOCALDOMS(n)$ to denote dominators within node n . In Figure 2(b), $LOCALDOMS(CHECK)=[check]$. For all other nodes, $LOCALDOMS$ has an empty value.

Stage 2. Propagating Dominators In this stage, we populate unconditional dominators up the call chain. This is achieved using a combination of a top-down intra-procedural traversal of the per node control flow graph and a bottom-up inter-procedural traversal of the global call graph. Intra-procedurally, a top-down traversal is performed to find the set of dominators generated in this node

that are unconditional, in other words, whether every path exiting the node includes the dominators. We use $GLOBALDOMS(node)$ to denote the set of unconditional dominators generated within the node $node$.

Inter-procedurally, a bottom-up traversal is performed to propagate the set of unconditional dominators up the call chain. Note that the two traversals (intra- and inter- procedural) are performed together. So as new dominators are propagated upward the call chain, the function $GLOBALDOMS$ gets updated along the way. Again take Figure 2(b) as an example, initially, $GLOBALDOMS(CHECK)$ has value `[check]` and for all other nodes the value is empty. Since node `f1` unconditionally calls the `CHECK` function, and thus $GLOBALDOMS(f1)$ is updated to value `[check]`. The final values of $GLOBALDOMS$ at completion of this stage are listed on the upper right hand side of each node in Figure 2. Figure 3 in the Appendix shows the pseudo code for propagating dominators.

Stage 3. Computing Dominators Globally

This stage is essentially the reverse of the second stage - we propagate the dominators computed in the second stage *down* the call graph. Similar to the second stage, this stage is also a combination of inter- and intra- procedural analysis. During the top-down traversal, which is an inter-procedural process, each node is visited and within each node an intra-procedural traversal of the control flow graph computes the dominator set for each outgoing call.

We use $e_{caller \rightarrow callee}$ to denote the edge from node $caller$ to node $callee$ in the call graph, $PRED(n)_i$ to denote the i th predecessor of node n , and $DOMINATORS(e_{caller \rightarrow callee})$ to denote the dominators that cover the edge $e_{caller \rightarrow callee}$. In addition, we expand the definition of $LOCALDOMS$ to cover edges. So $LOCALDOMS(e_{caller \rightarrow callee})$ means dominators generated within node $caller$ that dominates the edge $e_{caller \rightarrow callee}$. The function $DOMINATORS$ can thus be recursively defined as follows:

$$DOMINATORS(e_{caller \rightarrow callee}) = \bigcap_i DOMINATORS(e_{PRED(caller)_i \rightarrow caller}) \cup LOCALDOMS(e_{caller \rightarrow callee}). \quad (1)$$

Intuitively this means that the dominators covering edge $e_{caller \rightarrow callee}$ are the intersection of all dominators covering incoming edges into node $caller$ plus dominators generated inside node $caller$ that dominate edge $e_{caller \rightarrow callee}$. Again take Figure 2(b) as an example, the value $DOMINATORS(e_{f1 \rightarrow DOP})$, or

```

main() {
    struct secure_obj *o = getObject();

    f1(o);
    f2(o);
}

f1(struct secure_obj *o) {
    CHECK(o);
    DOP(o);
}

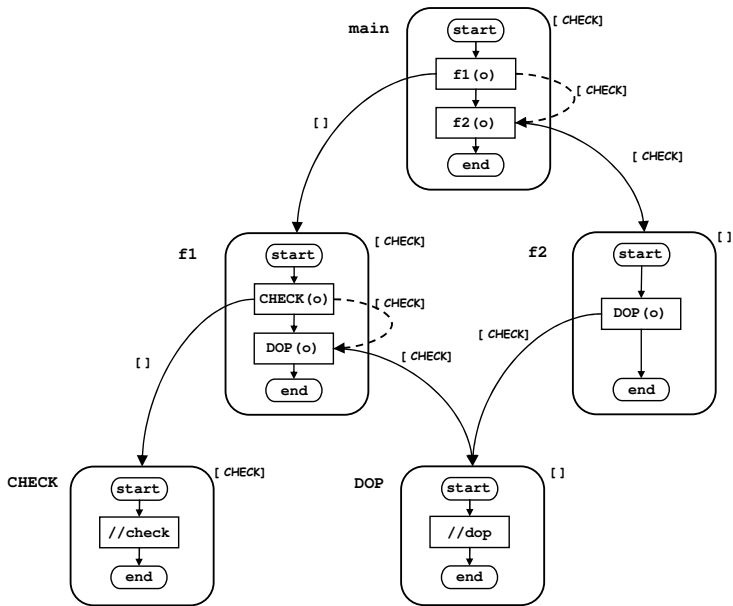
f2(struct secure_obj *o) {
    DOP(o);
}

CHECK(struct secure_obj *o) {
    // perform security check on o
}

DOP(struct secure_obj *o) {
    // perform dangerous operation on o
}

```

(a) Example Program.



(b) Super-graph Representing the Example Program.

Figure 2: Example Code and Its Super Graph.

the checks covering the edge from node $f1$ to node DOP is calculated by taking the intersection of the dominators for each incoming edge of node $f1$, in this case, $DOMINATORS(e_{main \rightarrow f1})$, and then merging the resulting set with internal dominators generated in node $f1$ that dominates the call to DOP , which in this case is the set $[check]$. This results in $DOMINATORS(E_{f1 \rightarrow DOP})$ having the value of $[check]$. The complete results for $DOMINATORS$ are shown as labels on each edge in Figure 2(b). Figure 4 in the appendix shows the pseudo code for computing this function.

3.2 Verifying Data Relationships Between the Dominator and the Dominatee

The dominator and dominatee form a data relationship that is part of the security specification. The LSM reference monitor interface (see Figure 1), for example, requires that the object being operated on (the file object Figure 1) is the same object being checked by the security hook. In some cases, the LSM interface requires the two data objects form a structure-field relationship where one data object is assigned to the field of the other. This latter check can be reduced to an equality comparison of one object and the field of the other object. Thus we focus our discussion on verifying equality of two data objects.

To verify the data relationships, the dominance algorithm is enhanced by associating the dominator/dominatee with the corresponding data object. Again take Figure 2(b) as an example, the dominator check will be attached with the object o which is passed to the $CHECK$ function as a parameter. The algorithm remains the same, except that when comparing two dominators (during intersection and merging operations in Equation (1)), the associated data objects will be compared as well.

Once the solution set for $DOMINATORS$ is computed for each edge, the analysis verifies that for each dominatee inside node n , the intersection of dominators covering each incoming edge plus dominators generated within node n that dominate the dominatee is not empty. We then compare the data object associated with the dominator with that of the dominatee and determine if the two data objects are the same.

To determine the equality of two data objects is a classical data flow problem. Given an analysis engine that supports sophisticated data flow analysis, this question can be answered *soundly*, meaning if the answer is yes then the two objects are definitely the same, without generating two many false positives. A *false positive* refers to the case that the algorithm gives a no answer, but the two data objects are actually the same.

In this paper, we use the data flow analysis engine provided by the JaBA framework [12]. JaBA represents the value of objects using *allocation sites*¹, defined as method and program point that performs the allocation of the object (such as a `malloc` call). The comparison can thus be done by comparing if two objects have the same allocation site. This will work if the object has only one allocation site. Take Figure 2(b) as an example, suppose the definition of `getObject()` is as follows:

```
1: static struct secure_obj *sec_obj =
2:   (struct secure_obj*)malloc(
3:     sizeof(struct secure_obj));
4:
5: struct secure_obj *getObject() {
6:   return sec_obj;
7: }
```

In this case both the parameter passed to `CHECK` and that passed to `DOP` refer to the same allocation site defined at line 2 by the `malloc` call. Thus we can safely conclude that the `CHECK` call dominates the `DOP` call with regard to their parameters `o`.

Unfortunately, for most programs of reasonable sizes, it is more common for an object to be associated with multiple allocation sites because of the object might take on different paths through the programs and thus pick up different values. In such cases, ambiguity arises even if the two objects have identical set of allocation sites, because one cannot be sure exactly which of the allocation sites the object refers to at the particular program point. Such ambiguity indicates possible race conditions between the dominator and the dominatee that might lead to TOCTTOU (Time of Check to Time of Use [4]) attacks. For example, the data object might refer to one of the allocation sites at the dominator program point and then might refer to another allocation site at the dominatee program point. Because we cannot be sure that the two objects are *always* the same, we cannot safely conclude that the domination relationship *always* holds. Again take Figure 2(b) as an example, suppose somewhere in the middle of the program another function `setObject()` is called which has the following definition:

```
7: void setObject() {
8:   sec_obj = (struct secure_obj*)malloc(
9:     sizeof(struct secure_obj));
10: }
```

The object `o` would then be associated with two allocation sites, the original allocation site plus the one from inside the call `setObject()`.

¹In this paper we use *value* and *allocation site* interchangeably.

Sophisticated analysis techniques such as context sensitivity might reduce such ambiguity, but cannot eliminate them all². For these remaining ambiguous cases, we employ a def-use chain analysis [1, 13] to help reduce ambiguity. A traditional def-use chain analysis tells us whether two objects have the same definition³. In Figure 2(b), the parameter of function `CHECK` and that of the function `DOP` share the same definition, namely, the assignment of the return value of function `getObject()` to the object `o`. We can thus safely conclude that the two parameters are the same⁴. This holds true even if somewhere between the dominator and the dominatee program points, the value of `sec_obj` is changed, because the value returned by `getObject()` stays the same.

Suppose however, that the definition of function `f2` is changed to the following:

```
11: void f2(struct secure_obj *o) {
12:   struct secure_obj *o2 = getObject();
13:   DOP(o2);
14: }
```

In this case, objects `o` and `o2` have the same set of allocation sites, but their definitions differ. The analysis thus concludes that the two objects are not the same, and that the `CHECK` call does not dominate the `DOP` call at line 13.

4 Results

4.1 Implementation

We implemented the dominance checking algorithm on top of the JaBA analysis framework⁵. JaBA implements inter-procedural control and data flow analysis. JaBA differs from most analysis tools in that it is context-sensitive, and thus is able to achieve high accuracy (low false positive rate). In addition, JaBA stores just enough context information so that it can scale up to hundreds of thousands of lines of code.

²otherwise we would have solved the halting problem.

³A definition is different from an allocation site in that a definition refers to an assignment to an object, whereas an allocation site refers to the original allocation point of the assigned value.

⁴Assuming that no alias of the memory location of parameter `o` exist that might change the value of `o`.

⁵However, JaBA is designed to analyze Java programs. To apply it to C code, we need to translate the C source code into Java. The translation is achieved using a source to source translation tool named FICTOJ (stands for Flow Insensitive C TO Java), which is covered in detail in another paper [16].

4.2 Results

The experiments were run on a 2.4GHz Mobil Pentium 4 IBM thinkpad T30 with 1GB of memory. To measure the accuracy of the tool, we compare the results with a previous approach based on CQUAL [15]. We analyzed Linux version 2.4.9 because the previous approach used this version.

Timing Performance. The Linux kernel is a large and complicated piece of software, with about 300 thousands of lines of C code (the portion that is actually compiled) and 4.2 MB in compiled bytecode. The LSM analysis took one hour and thirty four minutes to finish. We believe this is a great achievement because most other tools with similar levels of sophistication will not be able to analyze programs of this size at all.

Improving Accuracy. Our analysis tool based on JaBA generated 172 warnings, as compared to 524 with CQUAL. Thus we were able to reduce false positives by more than 60%. Note that a large amount of the positives are due to safe functions, and can not be eliminated by more accurate analysis. Thus the improvement is significant.

In summary, we have demonstrated that our dominance algorithm implementation on top of JaBA is scalable and yet still finds all errors that were captured using the CQUAL tool.

5 Conclusion

Systems with verifiable security properties provide users with a provable level of security assurance that has been lacking in most of today's software systems. In this paper we identify an important class of security properties that can be reduced to the classical *dominance problem*, and show how they can be checked efficiently and accurately, using a dominance algorithm implemented on top of the JaBA analysis framework. Our results demonstrate that our implementation can check the Linux kernel in less than two hours and yield significant fewer false positives compared to previous published results.

References

- [1] A. V. Aho, R. Sethi, and J. D. (Contributor) Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.
- [3] T. Ball and S. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, January 2002.
- [4] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.
- [5] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 4–6, 2004.
- [6] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, 2002.
- [7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*, October 2000.
- [8] J. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, 2002.
- [9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, June 2002.
- [10] ITSEC. *Common Criteria for Information Security Technology Evaluation*. ITSEC, 1999. Available at <http://csrc.nist.gov/cc/index.html>.
- [11] T. Jensen, D. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.
- [12] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 359–372, November 2002.
- [13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.
- [14] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [15] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [16] X. Zhang, T. Jaeger, L. Koved, and A. Kershenbaum. FICTOJ: A Flow Insensitive C to Java Translator, 2004. In Preparation.

Appendix

```

status := UNCHANGED;
queue := [ all nodes that have non-empty
          GLOBALDOMS value ];
function propagateDominators() {
  while (!queue.isEmpty()) {
    node := queue.pop();
    edges := node.getPredecessorEdges();
    foreach edge in edges {
      processEdge(edge);
      if (status == CHANGED) {
        predecessorNode = edge.getCaller();
        queue.push(predecessorNode);
      }
    }
  }
}

function processEdge(edge) {
  caller := edge.getCaller();
  method := caller.getMethod();
  basicBlocks := method.getBasicBlocks();
  // get the basic block that corresponds
  // to the call represented by edge
  dominatorBB := caller.getBasicBlock(edge);
  // the starting basic block
  queue := [ basicBlocks[0] ];
  // the set of reachable blocks
  reachableBasicBlocks := [ basicBlocks[0] ];
  while (!queue.isEmpty()) {
    basicBlock := queue.pop();
    succBasicBlocks :=
      basicBlock.getSuccBasicBlocks();
    foreach bb in succBasicBlocks {
      // terminate traversal at the
      // dominator basic block
      if ( bb != dominatorBB ) {
        queue.push(bb);
        reachableBasicBlocks.add(bb);
      }
    }
  }
  // if the exit basic block is not reachable,
  // then it means that all paths that return
  // from this method will include dominatorBB,
  // and thus this method contains an
  // unconditional path to the dominator.
  if ( !reachableBasicBlocks.
        contains(exitBasicBlock) ) {
    // update GLOBALDOMS function
    callee := edge.getCallee();
    GLOBALDOMS(caller).
      add(GLOBALDOMS(callee));
    status := CHANGED;
  }
}

```

Figure 3: Pseudocode for Dominator Propagation.

```

status := UNCHANGED;
queue := [ startNode ];
function computeDominators() {
  while (!queue.isEmpty()) {
    node := queue.pop();
    edges := node.getSuccessorEdges();
    foreach edge in edges {
      if (LOCALDOMS(edge) == null) {
        // we need to compute it
        computeLocalDominators(edge);
      }
      oldDominators := DOMINATORS(edge);
      // see equation (1)
      DOMINATORS(edge).union(
        LOCALDOMS(edge));
      predEdges := node.getPredecessorEdges();
      foreach predEdge in predEdges {
        DOMINATORS(edge).intersect(
          DOMINATORS(predEdge));
      }
      if (DOMINATORS(edge) != oldDominators) {
        // values have changed, need to update
        // successor nodes as well
        successorNode = edge.getCallee();
        queue.push(successorNode);
      }
    }
  }
}

function processEdge(edge) {
  caller := edge.getCaller();
  method := caller.getMethod();
  basicBlocks := method.getBasicBlocks();
  edgeBB := caller.getBasicBlock(edge);
  dominatorBBs := [ basic blocks that
                    contain/generate dominators ];
  foreach dominatorBB in dominatorBBs {
    // the starting basic block
    queue := [ basicBlocks[0] ];
    // the set of reachable blocks
    reachableBasicBlocks := [ basicBlocks[0] ];
    while (!queue.isEmpty()) {
      basicBlock := queue.pop();
      succBasicBlocks :=
        basicBlock.getSuccBasicBlocks();
      foreach bb in succBasicBlocks {
        // terminate traversal at the
        // dominator basic block
        if ( bb != dominatorBB ) {
          queue.push(bb);
          reachableBasicBlocks.add(bb);
        }
      }
    }
    // if the basic block that corresponds to the
    // edge call is not in the reachable set, then
    // it means that all paths that lead to that
    // basic block will include dominatorBB, and
    // thus the dominator dominates the edge.
    if ( !reachableBasicBlocks.
          contains(edgeBB) ) {
      // update GLOBALDOMS function
      LOCALDOMS(edge).add(dominator);
    }
  }
}

```

Figure 4: Pseudocode for Global Dominator Computation.