May 2003

Using Replication and Partitioning to Build Secure Distributed Systems

Lantian Zheng
Cornell University

Stephen Chong
Cornell University

Andrew C. Myers
Cornell University

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Copyright 2003 IEEE. Reprinted from *Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP 2003)* pages 236-250.
Publisher URL: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=27002&page=1

Using Replication and Partitioning to Build Secure Distributed Systems

Abstract

A challenging unsolved security problem is how to specify and enforce system-wide security policies; this problem is even more acute in distributed systems with mutual distrust. This paper describes a way to enforce policies for data confidentiality and integrity in such an environment. Programs annotated with security specifications are statically checked and then transformed by the compiler to run securely on a distributed system with untrusted hosts. The code and data of the computation are partitioned across the available hosts in accordance with the security specification. The key contribution is automatic replication of code and data to increase assurance of integrity—without harming confidentiality, and without placing undue trust in any host. The compiler automatically generates secure run-time protocols for communication among the replicated code partitions. Results are given from a prototype implementation applied to various distributed programs.

Using Replication and Partitioning to Build Secure Distributed Systems

Lantian Zheng Stephen Chong Andrew C. Myers
Computer Science Department
Cornell University
{zlt,schong,andru}@cs.cornell.edu

Steve Zdancewic
Dept. of Computer and Information Science
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

A challenging unsolved security problem is how to specify and enforce system-wide security policies; this problem is even more acute in distributed systems with mutual distrust. This paper describes a way to enforce policies for data confidentiality and integrity in such an environment. Programs annotated with security specifications are statically checked and then transformed by the compiler to run securely on a distributed system with untrusted hosts. The code and data of the computation are partitioned across the available hosts in accordance with the security specification. The key contribution is automatic replication of code and data to increase assurance of integrity—without harming confidentiality, and without placing undue trust in any host. The compiler automatically generates secure run-time protocols for communication among the replicated code partitions. Results are given from a prototype implementation applied to various distributed programs.

1 Introduction

Computing systems are becoming more complex and yet we increasingly depend on them to function correctly and securely. Unfortunately, it is currently difficult to make strong statements about the security provided by a computing system as a whole. Distributed systems make security assurance particularly difficult, as these systems naturally cross administrative and trust boundaries; typically, some of the participants in a distributed computation do not trust other participants or the computing software and hardware they provide. Systems meeting this description include clinical and financial information systems, business-to-business transactions, and joint military information systems. These systems are distributed precisely *because* they serve the interests of mutually distrusting principals.

The open question is how programmers should build distributed systems that properly enforce strong security policies for data confidentiality and integrity. In particular, we are interested in policies based on information flow (e.g., [16, 49, 57, 26]), which are attractive because they constrain the behavior of the whole system. Information flow policies are an *end-to-end* specification of computer security, unlike (discretionary) access control, which does not track information propagation.

Recently, *secure program partitioning* [61] has been proposed as a way to solve this problem. The Jif/split compiler automatically partitions high-level, non-distributed code into distributed subprograms that run securely on a collection of host machines that are trusted to varying degrees by the participating principals. (Such hosts are *heterogeneously trusted*.) A partitioning is secure if the security of a principal can be harmed only by the hosts the principal trusts. Thus, the partitioning of the source program is driven by a high-level specification of security policies and trust.

This work shows how to use replication to protect the integrity of program data and control information in the secure partitioning framework. Earlier work on secure partitioning found that integrity is a crucial aspect of system security, especially when trying to enforce complex, data-dependent security policies. Replication makes it easier to provide integrity because replicated data and computations can be checked against each other to ensure they agree. This is a well-known way to increase integrity assurance, used in file systems and replicated state machines (e.g., [40, 23, 7, 47]); what has not been previously investigated

This research was supported in part by DARPA Contract F30602-99-1-0533, monitored by USAF Rome Laboratory, in part by ONR Grant N00014-01-1-0968, in part by NSF Grant 0208642, and in part by an NSF CAREER award. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

is how to apply it to general computation in a system with mutual distrust.

Applying replication to secure program partitioning enables a broader class of applications to be automatically compiled for distributed systems with heterogeneously trusted hosts. But in this context, replication creates several new problems that this paper addresses:

- Trust is heterogeneous, unlike in the traditional applications of replication to fault tolerance. Therefore the replication of computation and data must vary from host to host, as determined by trust and security policies.
- Replication makes confidentiality policies *harder* to enforce, because it creates more copies of the data potentially vulnerable to attack. In our system, *secure hash replicas* are automatically generated to ensure integrity without violating confidentiality.
- For efficiency, replicated computation is performed concurrently. Therefore a suitable concurrency-control mechanism is needed for heterogeneously trusted hosts.

The rest of this paper is structured as follows. Section 2 gives some background on secure program partitioning, describing the security model and showing how programs are written using this model. Section 3 explains how programs are statically partitioned and replicated according to security constraints. Section 4 describes the run-time mechanisms that ensure the assumptions of the static analysis hold. The implementation of this approach and experience with it are discussed in Section 5, along with some performance results. Sections 6 and 7 conclude the paper with a discussion of related approaches, limitations of the existing system, and future work.

2 Programming and Security Models

2.1 Secure program partitioning

In the secure program partitioning approach, the desired computation is expressed as a non-distributed program containing security annotations. These annotations are used to check at compile time that the program does not contain disallowed information flows. The *splitter*, a back end to the compiler, also uses these program security annotations, along with information about the degree to which principals trust the available hosts, to construct a fine-grained secure partitioning of the program code and data onto these hosts.

Computations that would ordinarily be written as separate programs communicating over the network can be written as a single program; based on security considerations, the splitter automatically generates the separate subprograms and discovers a network protocol that they may use to communicate. The splitter operates automatically, but it may be given constraints and hints, for example to improve performance. It is not necessary to give the splitter the entire program at once; program code may be separately compiled. The use of explicit constraints and separate compilation can be useful for programmers who wish to partition their code by hand. In this usage the compiler and splitter serve to verify that the manual partitioning is secure.

An alternative approach to building secure distributed programs would be to start from a lower-level distributed program and add annotations that permit the program to be shown secure. However, secure program partitioning has some important advantages. First, the programmer need not be aware of the distributed protocols that are needed to ensure strong security properties. Second, it is not known how to annotate a program containing these complex protocols in a way that permits accurate determination of information flow. The closest existing work—on information flow in concurrent systems with a *trusted* execution platform [45, 52, 39, 6, 24, 19, 33]—has produced restrictive analyses that rule out many practical programs. We expect that dealing with mutual distrust would only exacerbate the problem. By contrast, secure program partitioning starts from a simpler, higher-level description of the computation and can be less restrictive because its security analysis has more information to work with.

This work is concerned with the control of information flow, including covert storage channels such as implicit flows [9]. We do not treat covert channels based on termination and timing, though some ongoing work partially addresses timing channels [1, 37]. In addition, our prototype implementation does not attempt to prevent certain traffic analysis attacks. Although all communication is suitably encrypted, an attacker could learn information based on the pattern of that communication. Adding dummy messages is one possible way to protect against these attacks [20, 2].

In this work the trusted computing base is the same as in the secure program partitioning work [61]; for example, it includes the Jif compiler and the splitter. However, a trusted host is not needed to perform the compilation and splitting of the program.

2.2 Security labels

The programs to be partitioned are written in the programming language Jif [29], which extends the Java language [48] with security annotations. Jif programs contain labels based on the *decentralized label model* [30], in which principals can express ownership of information-flow policies. This model works well for systems incorporating mutual distrust, because labels specify on whose behalf the security policy operates. In particular, label ownership is

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

used to control the use of *selective declassification* [34], a feature needed for realistic applications of information-flow control.

In this model, a *principal* is an entity (e.g., user, process) that can have a security concern. These concerns are expressed as labels, which state confidentiality or integrity policies that apply to the labeled data. Principals can be named in these policies as owners of policies and as readers of data.

A security label specifying confidentiality is written as $\{o{:}r_1,r_2,\ldots,r_n\}$, meaning that the labeled data is owned by principal o, and that o permits the data to be read by principals r_1 through r_n (and, implicitly, o). A label is a security policy controlling the uses of the data it labels; only the owner has the right to weaken this policy. A security label specifying integrity is written as $\{*{:}p_1,\ldots,p_n\}$, meaning that principals p_1 through p_n *trust* the data—they believe the data to be computed by the program as written. This is a weak notion of integrity; its purpose is to protect security-critical information from damage by subverted hosts. Labels combining integrity and confidentiality arise naturally; for example, the label $\{*{:}p_1;\ p_1{:}p_2\}$ indicates that the labeled data is trusted by principal p_1 and also owned by p_1, and only p_2 is permitted to read it.

Labels on data create restrictions on the use of that data. The use of high-confidentiality data is restricted to prevent information leaks, and the use of low-integrity data is restricted to prevent information corruption. The label on information may be securely changed from label L_1 to label L_2 if L_2 specifies at least as much confidentiality as L_1, and at most as much integrity as L_1. We write this label relationship as $L_1 \sqsubseteq L_2$. The relation $\sqsubseteq$ is a pre-order whose equivalence classes form a distributive lattice [30]; the lattice join and meet operations are $\sqcup$ and $\sqcap$ respectively. The join operation combines the restrictions on how data may be used. For example, if `x` has label L_x and `y` has label L_y, then `x + y` has label $L_x \sqcup L_y$, which preserves restrictions on the use of `x` and `y`. Dually, the label $L_x \sqcap L_y$ is at *most* as restrictive as L_x or L_y; thus, it must describe at least as much integrity as either label. This makes sense because labels represent restrictions on how data may be used; data with higher integrity has *fewer* restrictions on its use [4].

For any label (or program expression) x, the notations $C(x)$ and $I(x)$ refer respectively to the confidentiality and integrity parts of x (or the label of x).

2.3 The Jif programming language

Variables and expressions in Jif have types that may include security labels. For example, a value with type `int{o:r}` is an integer owned by principal `o` and readable by `r`. When unlabeled Java types are written, the label component is automatically inferred from the uses of the data.

Every program expression has a labeled type that is an upper bound (with respect to the $\sqsubseteq$ order) on the security of the data represented by the expression. Jif's type-checking algorithm prevents labeled information from being *downgraded*, or assigned a less-restrictive label (i.e., lower in the lattice). In general, downgrading results in a loss of confidentiality or a spurious increase in claimed integrity. The type system tracks data dependencies (information flows) to prevent unintentional downgrading.

Implicit flows [9] are information flows through the control structure of the program. Implicit flows can create both integrity and confidentiality concerns. For example, control-flow integrity is important: if untrusted parties can affect the control flow of the program, they might cause a security violation. Security policies on control flow are expressed as labels, just as for ordinary variables. For each program point *pc*, two special labels $C(pc)$ and $I(pc)$ are computed statically, representing the confidentiality and integrity policies applying to the control flow at that point.

A Jif programmer may annotate a program with arbitrary security labels, but this does not mean that the programmer has control over security. First, if labels are not internally consistent the program will not type-check; second, labels must also be consistent with the security policies on data in the external environment that the program interacts with. External consistency is checked partly at link time and partly at run time.

Strict information-flow policies based on noninterference [16] have not been successful in practice as they are very restrictive. Jif supports two operators for intentionally downgrading security policies, `declassify` and `endorse`. Declassification reduces confidentiality requirements and endorsement (dually) increases the claimed integrity of data. To downgrade a security label, code must be granted the authority of the principals whose security policies are affected. For declassification, these principals are the owners of the confidentiality policies that are weakened; for integrity, these are the principals newly appearing in the integrity part of the label.

A second restriction on downgrading is that it is only permitted at a point where the program counter label indicates that the affected principals trust program control flow. This requirement prevents the downgrading decision from being improperly affected by untrustworthy data or computation. The goal is to enforce the *robust declassification* property [59].

This work inherits some of the limitations of Jif. The most important is that programs are sequential; the Java `Thread` class is not available. This rules out an important class of timing channels whose control is an open research area [24, 44, 45].

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

```
1  int{Alice:; *:Alice,Bob} bid;
2  boolean{Alice:Bob; *:Alice,Bob} isCommitted;
3
4  void commit{Alice:Bob; *:Alice,Bob}
5         (int{Alice:; *:Alice} v)
6           where authority (Bob)
7  {
8    v = (v>=0) ? v : 0;
9    if (!isCommitted) {
10     bid = endorse(v, {*:Alice,Bob});
11     isCommitted = true;
12   }
13 }
14 int{Alice:Bob; *:Alice,Bob} reveal{*:Alice,Bob} ()
15     where authority (Alice)
16 {
17   if (isCommitted)
18     return declassify(bid, {Alice:Bob});
19   else return -1;
20 }
```

Figure 1. Bid commitment program

2.4 Bid commitment example

Figure 1 shows an example of a Jif program based on the well-known Bit Commitment Protocol [5]. Instead of committing a bit, the program commits a non-negative integer. The principal Alice commits a bid `v` to a principal Bob without revealing the bid. Later, Alice reveals `v` and Bob verifies that it is the bid Alice previously committed. We chose this example because it is short but has interesting security issues.

Alice's committed bid is represented by the field `bid`. Its label `{Alice:; *:Alice,Bob}` indicates that this field is owned (and can be read) only by Alice, and that both Alice and Bob trust it to be the committed bid. The boolean `isCommitted` records whether Alice has committed a bid yet; it must be trusted by both Alice and Bob and visible to both of them.

Lines 4 through 13 define a method `commit` that Alice uses to commit to the integer value `v`. Bob does not need to trust `v` because he does not care how Alice computes the value she commits to. The `endorse` operation makes that policy decision explicit—it boosts the integrity of the value of `v` so that it can be assigned to `bid`. The authority clause in line 5 gives the method Bob's authority, which is needed by the `endorse` operation. In lines 4 and 14, the label after the method name is a *start label*, used to control the implicit flow into the method [29].

Lines 14 through 20 define a method `reveal` that is used by Alice to reveal the committed bid to Bob. It returns the value of field `bid` if the value of `isCommitted` is true. If the value of `isCommitted` is false, it means that Alice has not committed a value yet, and `reveal` simply returns `-1`. Because Alice owns `bid`, releasing the data requires declassification and hence Alice's authority (declared at line 15) so that it can declassify Alice's data.

As shown in the example program, most security annotations that a programmer needs to specify are in method signatures. In general, programmers do not need to specify labels for local variables because they can be inferred automatically [29]. Usually, there are fewer security annotations in a program than type annotations, so writing down security annotations is not a much heavier burden than writing down the type annotations that programmers are used to.

2.5 Trust model and security assurance

Clearly, any secure distributed system relies on the trustworthiness of the underlying infrastructure. Let H be a set of *known hosts*, among which the program is to be distributed. We assume that pairwise communication between two members of H is authenticated, reliable, in-order, and cannot be intercepted or forged. Protection against interception and forgery can be achieved efficiently through well-known encryption techniques (e.g, [46, 58]).

To partition a program securely, the splitter must know the trust relationships between the participating principals and the hosts H. For example, if Alice declares that she trusts a host to hold her confidential data, the splitter can allow her data to reside on that host. Moreover, her confidentiality policy should be obeyed unless some host trusted by her suffers a malicious (Byzantine) failure, taking an action that is inconsistent with the subprogram located on h. Such an action might result from the subversion of h by an attacker. Conversely, a host that simply stops or crashes may cause the computation as a whole to halt, but should not harm data confidentiality or integrity.

Each host h has a security label that describes the trust that principals place in h. The confidentiality part of this label, $C(h)$, is an upper bound on the confidentiality of information that can be sent securely to h. The integrity part of the label, $I(h)$, is an upper bound on the integrity of information that can be received securely from h; that is, the set of principals that trust data from h. To authenticate a host label, each principal p needs to sign the security policies in the label that are owned by p.

The *trust configuration* is a map from all the hosts in H to their corresponding security labels. The splitter uses program labels and the trust configuration to securely partition a program. The partitioning must obey the constraint that the host selected to run a subprogram has a label that describes enough protection of confidentiality and integrity to execute that subprogram. A secure partitioning must satisfy the following security condition [61]:

> **Security Assurance:** Suppose H_{bad} is the set of compromised hosts in the system. Then the confidentiality of an expression e cannot be harmed

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

unless $C(e) \sqsubseteq \bigsqcup_{h \in H_{\text{bad}}} C(h)$; its integrity cannot be harmed unless $\bigsqcap_{h \in H_{\text{bad}}} I(h) \sqsubseteq I(e)$.

The intuition behind this condition is that the label of a host is a bound on the damage that the host can do if it is subverted. However, if multiple hosts are subverted, they may collude to cause more damage. Therefore, the damage caused by a set H_{bad} of compromised hosts should be bounded by the join of their confidentiality labels and the meet of their integrity labels.

The security assurance condition is not always easy to satisfy. Consider running the bid commitment program on a trust configuration in which host h_a's label is `{Alice:; *:Alice}` and host h_b's is `{Bob:; Alice:Bob; *:Bob}`. That is, h_a is trusted by Alice and can hold her private data, and h_b is trusted by Bob and can hold his private data as well as data Alice reveals to him. However, the original Jif/split system [61] cannot partition this code because the field `isCommitted` must be trusted by both Alice and Bob; therefore, the field cannot be placed on either h_a or h_b. This paper shows that replication of code and data can often solve this problem, which arises in realistic applications.

3 Partitioning and Replication

In this work the original secure program partitioning algorithm has been extended to exploit automatic replication. If there is no host with a sufficient integrity label to run a program statement or to store a field, the extended splitter can replicate the statement or the field on multiple hosts to satisfy integrity requirements.

Consider the partitioning failure described in Section 2.5. Unlike the original splitter, our extended Jif/split compiler can replicate the field `isCommitted` onto both h_a and h_b, so that the field's value is considered valid only when the copies on h_a and h_b agree. Alice trusts the copy on h_a and Bob trusts the copy on h_b; therefore, if both copies have the same value x, both Alice and Bob trust that the field `isCommitted` has the value x, as required by the field's integrity label, `{*:Alice,Bob}`. In general, by replicating data on a set of hosts $h_1, \ldots, h_n$, integrity may be increased up to the *combined integrity* $I(h_1) \sqcap \ldots \sqcap I(h_n)$ if the replicas all agree.

The use of replication increases the flexibility that the splitter has to partition programs, but the same security assurance condition still applies. Suppose e is replicated on a set of hosts h_i where $1 \leq i \leq n$. The splitter ensures statically that the combined integrity of the hosts h_i is sufficient to compute e, so $\bigsqcap_i I(h_i) \sqsubseteq I(e)$. The result of e can be incorrect only if all the replicas of e produce the same incorrect result; if so, the hosts h_i are all compromised, and we have $\bigsqcap_{h \in H_{\text{bad}}} I(h) \sqsubseteq \bigsqcap_i I(h_i)$. By transitivity, the result of e can be incorrect only when $\bigsqcap_{h \in H_{\text{bad}}} I(h) \sqsubseteq I(e)$, but then the security assurance condition does not guarantee the integrity of e.

Enforcing the integrity policies described here does not guarantee availability; if any of the hosts performing a replicated computation is compromised and produces a result inconsistent with the results from other hosts, the error will be detected and the computation will be halted. Better enforcement of availability policies appears to be possible within the secure partitioning framework, but is left to future work.

The rest of this section describes the replication and partitioning of classes and objects across a distributed system, as well as the static constraints that determine where each statement and each data item can be placed in the distributed system. These constraints ensure that confidentiality and integrity policies are enforced if all hosts compute correctly. Misbehaving hosts are controlled by the run-time mechanisms described in Section 4.

3.1 Splitting code, classes and objects

The splitter uses a fine-grained approach to partitioning. For each field and statement, the splitter assigns a set of hosts to it. Then statements and fields that can be placed on the same host are assembled to form a subprogram. This fine-grained approach gives the splitter flexibility in selecting hosts to satisfy the security constraints.

Like Java, Jif is an object-oriented language in which a program consists of classes. The splitter partitions a class into multiple *local classes*, each of which resides on one host. A local class contains some fields of the original class and stub code for calling class methods. If a class C is split into local classes $C_1, \ldots, C_n$, then an object o of class C is represented by a set of *local objects* $o_1, \ldots, o_n$ that are instances of the classes $C_1, \ldots, C_n$ and located on hosts $h_1, \ldots, h_n$. These local objects share the same *global object ID*.

The code of each source method is split into *code segments*. A code segment corresponds to a fragment of a source method and is identified by the source program point *pc* at which the fragment begins. Each code segment is replicated on a set of hosts; all the replicas simulate the computation of the source fragment.

A running method has an activation record that is explicitly represented as an object in the partitioned target code. Each activation record is partitioned into local *frame objects* that represent the part of the activation record that is stored on their host. As with local objects, local frame objects that represent the same activation record also share the same *global frame ID*.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

3.2 Selecting hosts for data

If a data item d is replicated on hosts $h_1, \ldots, h_n$, then each h_i must be trusted not to leak d to unauthorized readers. This constraint is expressed as $C(d) \sqsubseteq C(h_i)$ for all i, or equivalently, as $C(d) \sqsubseteq \bigsqcap_i C(h_i)$.

The hosts holding d may receive access or update requests for d from other hosts, and infer some information about the control flow. The splitter computes the confidentiality $C_{if}(d)$ of the implicit flow to each data item d: if d is accessed at a program point *pc*, the constraint $C(pc) \sqsubseteq C_{if}(d)$ is satisfied. The hosts h_i must be trusted to read the implicit flow: $C_{if}(d) \sqsubseteq \bigsqcap_i C(h_i)$.

The integrity of d is at most as high as the combined integrity of the set of hosts storing it: $\bigsqcap_i I(h_i) \sqsubseteq I(d)$. Thus, replicating d tends to make it easier to satisfy the integrity constraint but harder to satisfy the confidentiality constraints: there is a tension between confidentiality and integrity. One way to resolve this tension is to store a secure hash value of d on hosts that cannot read d. The user of d can verify the real value of d against its hash value to assure integrity. We refer to the hashed copies of a piece of data as its *hash replicas*. Confounders are used to protect hash replicas against dictionary attacks, as described in Section 4.4.1.

Suppose a host h holds a hash replica of d. While it cannot determine the real value of d, it knows when d is accessed. Therefore h must have a confidentiality label at least as high as $C_{if}(d)$. For hash replicas, there are three constraints for placing d on hosts $h_1, \ldots, h_n$:

$$\begin{array}{l} \exists_i\, C(d) \sqsubseteq C(h_i) \\ C_{if}(d) \sqsubseteq C(h_1) \sqcap \ldots \sqcap C(h_n) \\ I(h_1) \sqcap \ldots \sqcap I(h_n) \sqsubseteq I(d) \end{array}$$

The first constraint ensures that there exists at least one host that can hold v's real value. The second constraint ensures that the hosts holding the data are trusted to receive the implicit flows. The third says that collectively the set of hosts satisfy d's integrity requirement.

Consider the field `bid` of Figure 1. It has the label `{Alice:; *:Alice,Bob}`, and C_{if}(`bid`) is `{Alice:Bob}` because the value of `isCommitted` can be inferred from the fact that `bid` is updated at line 9. Thus, `bid` is replicated on h_a and h_b, and h_a can hold its real value while h_b can only hold its hash. It is easy to check that the three constraints are satisfied: C(`bid`) $\sqsubseteq C(h_a)$, C_{if}(`bid`) $\sqsubseteq C(h_a) \sqcap C(h_b)$ and $I(h_a) \sqcap I(h_b) \sqsubseteq I$(`bid`).

3.3 Selecting hosts for code

In general, the hosts running a statement need to read all the inputs of the statement. However, knowing hash replicas of some inputs is sufficient for execution of some common statements such as assignments. Consider the statement `bid=endorse(v,{*:Alice, Bob})` in the bid commitment example. This statement is translated into `bid=v` and replicated on h_a and h_b. Since h_b is not allowed to read the real value of `bid` or `v`, it owns hash replicas of both `bid` and `v`. To execute the statement, h_b only needs to assign the hash replica of `v` to the hash replica of `bid`—no computation that depends on the actual value of `v` takes place. Given a statement s, let $U_r(s)$ be the set of inputs whose real values are needed in the computation of s, and let $U_h(s)$ be the set of inputs whose hash replicas are sufficient to carry out s. Then $C(s) = \bigsqcup_{v \in U_r(s)} C(v)$.

The hosts running a statement s also need to have a combined integrity at least as high as the integrity of any output of s. Let $D(s)$ be the set of locations s defines. Then $I(s) = \bigsqcap_{l \in D(s)} I(l)$. In general, hosts $h_1, \ldots, h_n$ can execute the statement s securely if the following three constraints are satisfied:

$$\begin{array}{l} \forall_{v' \in U_h(s)}\, \exists_i\, C(v') \sqsubseteq C(h_i) \\ C(s) \sqsubseteq C(h_1) \sqcap \ldots \sqcap C(h_n) \\ I(h_1) \sqcap \ldots \sqcap I(h_n) \sqsubseteq I(s) \end{array}$$

The first constraint guarantees that there exists at least one host h_i that can hold the real value of the input v'. The second constraint requires that every host can read those inputs whose real values are needed to execute s. The third constraint ensures that the set of hosts has a combined integrity sufficient for every output of s.

4 Run-time Mechanisms

As a partitioned program runs, code segments on different hosts interact to simulate the control flow and data flow of the source program as if it were running on a single machine. These interactions include control and data transfers between hosts, both of which are supported by the run-time system. Each call to the run-time system sends a message to another host to trigger an action on that host, such as executing a code segment or accessing a field. An important goal of the run-time system is to prevent bad hosts from causing integrity violations.

4.1 Run-time interface

Figure 2 shows the interface to the run-time system. There are three operations for transferring data between hosts. Calls to `getField` and `setField` access remote fields, while `forward` transfers local variables between frame objects on different hosts. The other three operations in the figure—`rgoto`, `lgoto` and `sync`—are used to transfer control among the hosts. This run-time interface is similar to that in the original Jif/split system [61], but its

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

```
Val  getField(Host h, Obj o, Field f)
void setField(Host h, Obj o, Field f, Val v)
void forward(Host h, FrameID f, Var var, Val v)
void rgoto(Host h, FrameID f, int pc)
void lgoto()
void sync(FrameID f, int pc)
```

Figure 2. Run-time interface

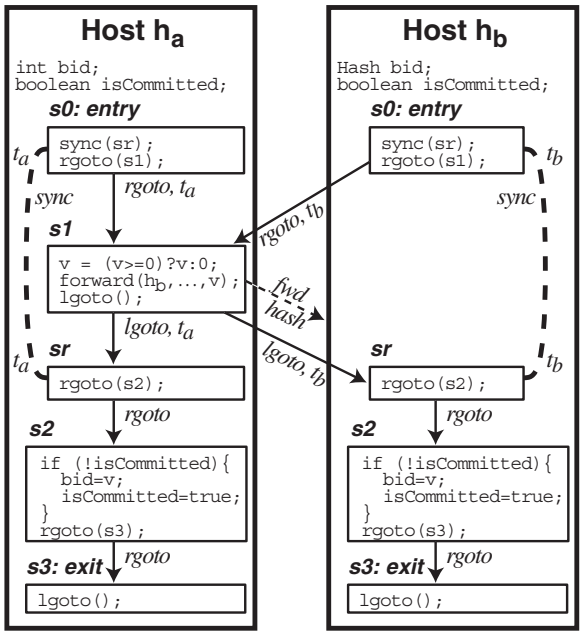


Figure 3. Control flow graph of the `commit` method

implementation is quite different because it may be used concurrently by different replicas.

Both `rgoto` ("regular goto") and `lgoto` ("linear goto" [60]) operations transfer control to a code segment on a remote host. Intuitively, `rgoto` is used to transfer control from a code segment to another with equal or lower integrity[1], while `lgoto` allows a code segment to transfer control to another code segment with higher integrity. Figure 3 shows the control-flow graph of a possible splitting and replication of the `commit` method in Figure 1, and illustrates how `rgoto`, `lgoto` and `sync` are used to transfer control. In Figure 3, the integrity labels of s_0, s_1 and s_2 are respectively `{*:Alice,Bob}`, `{*:Alice}` and `{*:Alice,Bob}`.

- `rgoto`(h, f, pc) invokes the code segment at *pc* on host h, with frame object f. The hosts doing the `rgoto` must have a combined integrity as high as that of the code segment to be invoked. In Figure 3, s_0 transfers control to s_1 with `rgoto`.

[1]The integrity label of a code segment s is the meet of the integrity labels of all the statements in s.

- `lgoto`() transfers control from one code segment to another with higher integrity. A capability mechanism prevents a host from using an invalid `lgoto` to corrupt a computation with higher integrity. In Figure 3, after running s_1, host h_a sends two `lgoto` requests to invoke the two replicas of s_r. Since the integrity of h_a is lower than that of s_r, h_a must present a capability for invoking s_r. Unlike in the original Jif/split system [61], the capability is a *set* of capability tokens $\{t_a, t_b\}$. Each token is used to invoke a replica of s_r.
- `sync`(f, pc) creates a capability token t that can be used to invoke the code segment replica on the local host with frame object f. In general, a `sync` operation is replicated on multiple hosts, and creates a set of tokens. A capability token is a tuple $\langle h, f, pc, uid \rangle$, containing a host ID, a frame ID, a program counter, and a unique 128-bit identifier. The first three components specify the code segment to be invoked by the token. The last component prevents forgery and ensures uniqueness with high probability. In Figure 3, the replicas of s_0 do `sync` operations to collectively generate the capability $\{t_a, t_b\}$.

4.2 Replication and run-time checks

Except for `sync`, all of the operations in the run-time interface need to send a message to another host. This has two security implications. First, the receiving host must protect the confidentiality of the message, and second, a message cannot be trusted more than its sender.

Suppose a run-time call on host h sends a message m to host h' to invoke an action a. Let $C(m)$ be the confidentiality of the information that is contained in m or can be inferred from it, and let $I(a)$ be the integrity required to perform a. Then the system must enforce two security constraints: $C(m) \sqsubseteq C(h')$ and $I(h) \sqsubseteq I(a)$. The splitter statically ensures that $C(m) \sqsubseteq C(h')$ when it generates the code for the run-time call. However, the condition $I(h) \sqsubseteq I(a)$ must be checked at run time, because bad hosts might fabricate messages.

The constraint $I(h) \sqsubseteq I(a)$ has an interesting interaction with replication. Suppose a statement `f=3` is replicated on hosts h_1 and h_2, and the field `f` resides on host h'. On executing the statement, both h_1 and h_2 send a `setField` message to h'. Host h' should update `f` if the combined integrity of h_1 and h_2 is as high as that of `f`. Suppose h' receives the `setField` message from h_1 first and finds that $I(h_1) \not\sqsubseteq I(\texttt{f})$. In that case, h' suspends the request until the same request is made by h_2. Then, h' accepts the request after verifying that $I(h_1) \sqcap I(h_2) \sqsubseteq I(\texttt{f})$. In general, if $h_1, ..., h_n$ request an action a, it can be performed securely if the combined integrity of the hosts h_i is sufficient:

$$I(h_1) \sqcap ... \sqcap I(h_n) \sqsubseteq I(a) \qquad \text{(RC1)}$$

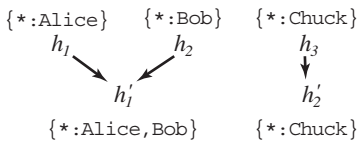
Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

However, this condition is more restrictive than necessary. In general, if hosts $h_1, \ldots, h_n$ send messages to hosts $h'_1, \ldots, h'_r$ to invoke an action a, each h'_j can securely do a if the following condition holds:

$$I(h_1) \sqcap \ldots \sqcap I(h_n) \sqsubseteq I(a) \sqcup I(h'_j) \qquad \text{(RC2)}$$

The action a is successfully performed only if all the hosts h'_j perform it, implying that RC2 holds at all h'_j. But this implies $\prod_i I(h_i) \sqsubseteq I(a) \sqcup \prod_j I(h'_j)$, which guarantees that RC1 is satisfied because the splitter statically ensures $\prod_j I(h'_j) \sqsubseteq I(a)$. Therefore, it is safe to use RC2 in run-time checks.

Suppose hosts h_1, h_2 and h_3 want to update a field `f` that is replicated on hosts h'_1 and h'_2. The integrity of `f` is `{*:Alice,Bob,Chuck}`, and the integrity labels of the five hosts are shown in the following figure. Using RC2, only three messages are required:



4.3 Control transfer mechanisms

Using the three run-time calls (`rgoto`, `lgoto` and `sync`) as building blocks, the splitter generates the run-time protocol that simulates the control flow of the source program. A secure control transfer protocol must prevent low-integrity hosts from affecting high-integrity control flow. Otherwise, a bad low-integrity host may compromise high-integrity computation, leading to data corruption or improper declassification of confidential data.

The difficult case is when the control needs to be transferred from a code segment s_1 to another segment s_2 with higher integrity ($I(s_1) \not\sqsubseteq I(s_2)$). This transfer is potentially insecure, because the bad hosts may have sufficient integrity to invoke s_1 and cause control to be passed to s_2 even though they do not have enough integrity to invoke s_2. So s_1 must use an `lgoto` operation along with a set of capability tokens to invoke s_2.

4.3.1 The `lgoto` protocol

Consider a simple control flow $s_0 \rightarrow s_1 \rightarrow s_2$, where s_0 and s_2 have higher integrity than s_1. Control should go from code segment s_0 to s_1, and then to s_2. Here, s_1 has low integrity and cannot pass control to s_2 directly, but s_0 has sufficient integrity to transfer control to s_2. So when s_0 passes control to s_1, it gives s_1 a capability that permits control to be returned to a segment s_r containing the statement `rgoto`(s_2). The segment s_r has the same integrity as s_0 and resides on the same set of hosts. Intuitively, s_1 is like a procedure call, and s_r is the return address.

Suppose s_0 is replicated on hosts h_1 through h_n. The protocol works roughly like this: each replica of s_0 on h_i does a `sync` operation to create a token t_i for the replica of s_r on the same host. Then s_0 passes control and the set of tokens $t_1, \ldots, t_n$ to s_1. After s_1 finishes running, it returns control to s_r with those tokens. Finally, s_r runs and transfers control to s_2. Using the notation "$s_0 \rightarrow s_1 : m$" to represent sending a message m from s_0 to s_1, we can write the protocol as:

1. s_0 : `sync`(s_r), creates $t_1, \ldots, t_n$
2. $s_0 \rightarrow s_1$: `rgoto`, $\langle t_1, \ldots, t_n \rangle$
3. $s_1 \rightarrow s_r$: `lgoto`, $\langle t_1, \ldots, t_n \rangle$
4. $s_r \rightarrow s_2$: `rgoto`

This description hides some complexity arising from replication. In step 2, sending an `rgoto` request from s_0 to s_1 actually requires multiple network messages from the hosts running s_0 to the hosts running s_1. In step 3, if a host running s_1 has the token t_i, the host just sends t_i along with an `lgoto` request to h_i. Figure 3 shows an example in which s_0 and s_1 are replicated on two hosts.

This protocol handles a simple control transfer to and from a low-integrity host; more complex control flow can always be reduced to occurrences of this simple case.

4.3.2 The `rgoto` protocol

Suppose code segment s_0 transfers control to segment s_1 with an `rgoto`, where s_0 is replicated on $h_1, \ldots, h_n$ and s_1 is replicated on $h'_1, \ldots, h'_m$. For each h'_j, the splitter finds the smallest subset of $\{h_1, \ldots, h_n\}$ such that the combined integrity of the subset of hosts is greater than or equal to $I(h'_j) \sqcup I(s_1)$; every host in the subset sends an `rgoto` request to h'_j. This protocol guarantees that the run-time check of RC2 at each host h'_j succeeds, and avoids unnecessary network communication.

4.3.3 Token management

Capability tokens allow low-integrity hosts to invoke high-integrity code segments using `lgoto`, so it is critical to restrict the creation and propagation of these tokens. Suppose that I is an arbitrary integrity label. A label I' is considered high-integrity if $I' \sqsubseteq I$; otherwise it is low-integrity. To prevent misuse of capability tokens, the system must enforce two security invariants for every integrity label I:

TI1 When control is in a high-integrity code segment, no set of hosts whose combined integrity is low has a complete set of tokens for a high-integrity code segment.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

TI2 When control is in a low-integrity code segment, at most one complete set of tokens for a high-integrity code segment is held by any set of hosts whose combined integrity is low.

TI1 prevents low-integrity hosts from starting a high-integrity thread while one is already running. TI2 ensures that once control is transferred to a low-integrity code segment, high-integrity control flow can be resumed at only one point. These two invariants leave low-integrity hosts no choice but to follow the control flow chosen by high-integrity hosts.

In the control-transfer protocol, tokens are passed between hosts in two ways. First, tokens can be passed along with `lgoto` requests, as shown in step 3 of the `lgoto` protocol. The run-time system of each host maintains an *entry table* that records tokens created on that host and their corresponding code segments. When a host receives an `lgoto` request with a token t, it checks the entry table. If t is in the entry table, the host invokes the corresponding code segment, and deletes t from the entry table to prevent replay attacks.

Second, tokens can be passed along with `rgoto` requests, as shown in step 2 of the `lgoto` protocol. These tokens can be used by the destination hosts to invoke a remote code segment. The run-time system associates the tokens received along with an `rgoto` request with the code segment invoked by the `rgoto` request. The tokens associated with the running code segment are called the *active token set* (ATS). For instance, in Figure 3, the ATS of s_1 is $\{t_a, t_b\}$. Suppose host h is running code segment s. Depending on what control transfer operations are performed by s, the run-time system of h manages the ATS of s in one of three ways:

- Case 1: `lgoto`. The ATS is used to return control to some higher-integrity code segment. Recall that an `lgoto` call does not have any arguments, because the run-time system maintains the ATS.
- Case 2: `rgoto`. The ATS is distributed to the replicas of the destination code segment along with the `rgoto` request.
- Case 3: `sync` followed by `rgoto`. The `sync` call creates a new token t that corresponds to a code segment s_r, which should be the only return point for the following computations that has a lower integrity than s. Token t is sent along with the `rgoto` request, and the current ATS becomes associated with s_r, so that it becomes the ATS when control returns.

There is an important security constraint about the distribution of tokens in case two. Suppose a set of hosts use `rgoto` to transfer control to a code segment s replicated on $h'_1, \ldots, h'_m$, and distributes a set of tokens to each of the replicas. Then any subset B of $\{h'_1, \ldots, h'_m\}$ with $I(B) \not\sqsubseteq I(s)$ cannot receive the complete set of tokens. This is a direct corollary of TI1, where high integrity and low integrity are defined with respect to $I(s)$. If B receives the complete set of tokens, then TI1 is violated: the control is in a high-integrity code segment s, but B has a low combined integrity label and holds a complete set of tokens for a high-integrity code segment.

The run-time system enforces this constraint by ensuring that each recipient gets at least one unique token. However, the senders may not have enough tokens to assign a unique one to each recipient. In that case, the run-time system splits a token into multiple tokens by a secret-splitting scheme based on the exclusive-or operation [42].

4.3.4 Control flow assurance

The control flow assurance that our control transfer mechanism is designed to enforce can be defined using a trace model. We represent an execution of a program as a trace of code segments that are running sequentially. For example, $F = s_0 s_1 s_r s_2$ is a trace. Let $\lfloor F \rfloor_I$ represents the trace obtained by removing from F those code segments with low integrity relative to I. Intuitively, $\lfloor F \rfloor_I$ should not be corrupted by a set of bad hosts whose combined integrity is lower than I. The control transfer mechanism is intended to enforce the following property:

> **Control Flow Assurance** Let F be the correct trace of running a program, F' be the actual trace of running the same program, and H_{bad} be the set of compromised hosts. Then $\bigsqcap_{h \in H_{\text{bad}}} I(h) \not\sqsubseteq I$ implies $\lfloor F' \rfloor_I$ is a prefix of $\lfloor F \rfloor_I$.

Recall that bad hosts can potentially stop the computation. That is the reason why the condition states that $\lfloor F' \rfloor_I$ is a prefix of $\lfloor F \rfloor_I$ instead of equal to $\lfloor F \rfloor_I$. However, this availability attack will not corrupt data or cause confidential data to be leaked.

Our control transfer mechanism ensures that low-integrity hosts can only use capability tokens to invoke a high-integrity code segment. Under this condition, the two token invariants imply the control flow assurance.

4.4 Data transfer mechanisms

Data transfer operations include accessing fields, updating fields, and forwarding local variables. To read a field `f`, a host h sends `getField` requests to a host set $H_{\mathtt{f}}$ that hold `f` and have a combined integrity as high as $I(\mathtt{f}) \sqcup I(h)$. Each host in $H_{\mathtt{f}}$ returns the value of `f` to h after checking that $C(\mathtt{f}) \sqsubseteq C(h)$. Then h compares the replicas of `f` from $H_{\mathtt{f}}$, and accepts the value only if all the replicas are the same.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

To update a field `f` replicated on h'_1 through h'_n, the updating hosts send `setField` requests to each h'_i, which do the update after checking RC2. If a running code segment updates a local variable, it has to forward the update to other code segments residing on remote hosts that may use the variable.

4.4.1 Data hashing

As described in Section 3.2, a secure hash value of data d may be stored on a host h whose confidentiality is only as high as $C_{if}(d)$. The run-time system uses the MD5 algorithm [36] to generate the hash. If host h wants to create a hash of data d, it generates a confounder n and computes $\{d, n\}_{\text{MD5}}$. Whenever h sends d to some host, it also sends n to that host so that the recipient can verify that $\{d, n\}_{\text{MD5}}$ is the hash of d. If d is replicated on multiple hosts, those hosts have to create the same confounder for d. The run-time system uses the global identifier generation algorithm of Section 4.5 to generate shared confounders.

In Figure 3, the code segment s_1 contains a statement `v=(v>=0)?v:0` that defines `v`, and s_2 contains a statement `bid=v` that uses `v`. After running s_1, h_a needs to forward the value of v to the replica of s_2 on h_b. Since h_b cannot read v, h_a only sends the hash value of `v` to h_b. It is interesting that the usual way of implementing the bit commitment protocol is to have Alice send a hash value of her committed bit to Bob. The splitter automatically generates a similar protocol from the high-level security policy.

4.4.2 Data consistency and synchronization

Several hosts may run the same piece of code or access the same data concurrently. To maintain consistency, the run-time system must ensure that those accesses are properly ordered. Suppose a field is replicated on a set of hosts. It is important that each host processes the `getField` and `setField` requests in the order specified by the source program. However, requests are generated by replicated code segments that need not be synchronized with one another. A host should not serve a request until all logically previous requests have been served. Timestamps are a common way to accomplish this, but timestamps may leak confidential information about control flow. Instead, the hosts storing field replicas coordinate with each other using the following protocol.

A host receiving a new access request acts as the coordinator of a two-phase commit protocol that ensures all other replica hosts are aware of the request. It announces the existence of the request to the other replica hosts, which acknowledge the announcement. Once all acknowledgements have been received, the request is serviced by the coordinator, and in parallel it informs the other replicas, permitting them to begin servicing that request as well. Every host delays servicing a field request until it has served all pending requests that it has acknowledged.

Some simple optimizations reduce the number of messages sent. A read request may be serviced if all pending requests are also reads. In general several hosts may receive a new request concurrently and each try to act as coordinator. However, coordinators are arbitrarily ordered and a host stops participating in a run of this protocol once it becomes aware of a different run for the same request with a lower-numbered coordinator. Finally, if there are only two replicas, the final step of the two-phase commit is skipped because it is not necessary.

4.5 Global identifier generation

Both object IDs and frame IDs are global and must be generated consistently by replicated code segments. However, care must be taken to avoid creating a covert information channel in which information about the control flow on trusted hosts is deducible from the global identifier. In our implementation, the covert channel is avoided by making an identifier appear random to hosts other than the creators. Every set of hosts that may create a global identifier share a secret confounder, which is used to generate global identifiers independently and efficiently using MD5 hashing. The identifiers created with a confounder appear random to hosts who do not know the confounder. At the start of the program, hosts need to create global IDs run an agreement protocol to initialize the confounder.

5 Results

The splitter and the necessary run-time support for executing partitioned programs has been implemented in Java as an extension to the existing Jif compiler [31].

5.1 Benchmark Programs

The system was evaluated with a set of programs that explored different kinds of distributed protocols and security configurations; these programs were also compared with hand-coded implementations.

Based on previous experience, communication cost is the greatest contributor to execution time in a WAN environment. Since the distributed system will typically cross administrative boundaries, we expect a WAN environment to be the norm, and therefore report performance in terms of the number of host-to-host messages generated.

The execution of the partitioned programs on different hosts was simulated with multiple threads in a single JVM, and the number of messages between hosts counted. Each host's subprogram was executed in a different thread.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

The hand-coded implementations were also run on multiple threads in a single JVM.

The benchmark programs used were auctions, a banking simulation, and the game Battleship. Replication of both code and data was required to successfully partition these programs with the trust configurations used. The programs used are fairly short but contain the same security issues that would be found in a more complete implementation.

In our most full-fledged example, Battleship, there are 44 security annotations (labels), which is approximately 1 annotation for every 3 lines of code.

To summarize the results, the run-time performance of the system on these programs was reasonable, and replication allows us to successfully partition programs for a larger class of trust configurations.

5.1.1 Auctions

Auctions are a useful component of a number of electronic commerce interactions [21], for example electronic procurement, where suppliers are bidding to fulfill a contract. Participants in these interactions may have confidentiality and trust requirements on the information used and exchanged; various types of auctions, such as closed bid auctions, incorporate aspects of privacy and trust. The diverse security requirements of different types of auctions, and their relevance to electronic commerce, make auctions an interesting and suitable problem area.

Three different types of auctions were modeled. All the auctions modeled are one-sided, first price auctions, with only a single item for sale. The seller and the bidders are identified with principals. Due to the single-threaded nature of the programs, bids cannot be submitted asynchronously. Instead, a round of bidding consists of each bidder in turn submitting a bid.

The three auctions are named A_1, A_2 and A_3. Auction A_1 is an open bid auction—all bids are public, and are endorsed by all principals. At the close of the bidding, computation of the winning bidder is performed publicly. Auction A_2 is a sealed bid auction, where bids are made public at the close of bidding. Auction A_3 is similar to A_2 except that bids are revealed only to the seller, who then determines the winning bidder and reveals the result. The privacy and integrity requirements of each of these auctions are expressed in their programs as labels and uses of downgrading.

5.1.2 Banking Simulation

Banking is an important distributed application with complicated privacy and integrity requirements. We implemented a simple banking example: Alice holds a credit-card with a bank, and two credit report agencies maintain a credit report for her. If Alice pays her credit-card bill late, the bank reports this to the agencies, and then asks the agencies for Alice's credit rating. If the rating is too low, the bank may cancel her line of credit.

We model Alice's bank account information as being owned by Alice, readable by the bank, and trusted by both of them. For the bank to send a report to the agencies thus requires an explicit declassification of information by Alice, which is presumably authorized by Alice when she opens the account. Alice's credit report has the label `{A:C1,C2;*:C1,C2}`, where `A` represents Alice and `C1` and `C2` represent the credit report agencies. To achieve the required security assurance, the code and data for the credit report must be replicated on the hosts of both agencies.

5.1.3 Battleship Game

Battleship is a game for two players. Each player has a secret grid containing several battleships. In turn, each player asks the opponent to reveal the contents of a particular location on the opponent's grid. Play continues until one player wins by discovering the location of all of his opponent's battleships.

This simple game has a number of interesting security properties. Each principal (player) has a grid that is readable only by its owner, but to prevent cheating it must be trusted by both principals. The principals alternate between testing a single location of the opponent's grid for a ship, and declassifying that information. The control flow of the program must be trusted by both principals, to ensure that turns alternate strictly and thus that no principal reveals too much at once. At the end of the game, the unrevealed portion of each principal's grid is declassified, to verify that both principals had the same number of battleships.

Many of the security issues that arise in this simple game, such as preventing client cheating, are relevant to more realistic online gaming systems. We speculate that program partitioning may be useful for constructing secure online games.

5.2 Performance

All the auction scenarios were run with three bidders. In auctions A_1 and A_2 the seller plays no role in the bidding process or in the computation of the winning bid, and is not explicitly represented. The banking program BNK was run with 4 principals, representing the bank customer, the bank and two credit reporting agencies. The Battleship program BTL was run with two principals on a 10×10 grid.

Table 1 shows the trust configurations that were used for the experiments. Each column shows the host labels occurring in one configuration. For the auctions, principals `A`, `B`, and `C` are bidders and `S` represents the seller; for Banking, principal `A` is the bank customer, `B` represents the bank, and `C1` and `C2` represent credit reporting agencies; for Battleship, principals `A` and `B` are the two players.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

Host	Config X	Config Y	Config Z	Config B	Config W
h_1	{A:; *:A,B}	{A:; S:A; *:A}	{A:; S:; *:A}	{A:B; *:B,C1,C2}	{A:; *:A}
h_2	{B:; *:B,C}	{B:; S:B; *:B}	{B:; S:; *:B}	{A:C1; *:C1}	{B:; *:B}
h_3	{C:; *:C}	{C:; S:C; *:C}	{C:; S:; *:C}	{A:C2; *:C2}	–
h_4	–	{S:; *:A,B,C,S}	{S:; *:S}	–	–

Table 1. Trust configurations for example programs.

Metric	A_1(hA_1)	A_2(hA_2)	A_3(hA_{3Y})	A_3(hA_{3Z})	BNK(hBNK)	BTL(hBTL)
Lines	49 (78)	54 (85)	62 (94)	62 (175)	53 (120)	142 (162)
Configuration	X	X	Y	Z	B	W
Total messages	11 (4)	24 (8)	47 (9)	27 (18)	16(8)	1294 (383)
`forward`	1	7	9	9	4	1109
`lgoto`	2	2	9	0	2	0
`rgoto`	8	15	29	18	10	185

Table 2. Program measurements

The first row of Table 2 gives the program lengths in lines of code. We measured total message counts using configuration X for A_1 and A_2, configurations Y and Z for A_3, configuration B for Banking and configuration W for BTL (as shown in the next row). The subsequent rows give total message counts and a breakdown of counts by type for the automatically partitioned program.

No `setField` or `getField` messages were sent during any of the simulations—all field accesses were local. The splitter is often able to avoid `setField` and `getField` calls because of the increased spatial locality of the data that results from replication: because fields are replicated on multiple hosts, a host can often access its local replicated copy, instead of communicating with other hosts.

The results from the hand-coded implementations of the example programs are shown in the table in parentheses (hA_1, hA_2, hA_{3Y}, hA_{3Z}, hBNK and hBTL of Table 2). The hand-coded implementations provide the same security assurance as the automatically partitioned programs, and explicitly replicate data and code to achieve the required integrity. The insight obtained by reading the corresponding partitioned code helped in writing the reference implementations securely and efficiently.

All of the hand-coded implementations are longer than the corresponding automatically partitioned programs. Also, the hand-coded implementations were written for specific trust configurations; hA_{3Y} and hA_{3Z} were both coded from scratch, while A_3 was recompiled with different trust configurations. In general, partitioning a program for different trust configurations is very easy; it is simply a matter of recompilation.

The hand-coded implementations send 1.5–6× fewer messages than the automatically partitioned programs. This efficiency is possible because the hand-coded programs exploit concurrency to a greater degree than our automatically partitioned programs. Our system must be conservative in its use of concurrency to ensure that the security protocol of a program—often implicit in the sequencing of execution—is adhered to. For example, in the bid commitment program of Figure 1 there is an implicit synchronization point after the `endorse` statement of the `commit` method, to ensure that Alice has really committed to a value before computation proceeds. Our system thus conservatively follows the flow of control of the source program, while the hand-coded programs are free to rearrange control flow so long as the security constraints are met. However, as program complexity increases, writing secure concurrent code by hand, and guaranteeing its security, can be difficult. Our system trades off some of the expressiveness and performance of hand-written concurrent code for the assurance that the principals' security policies are adhered to.

5.3 Discussion

If our new Jif/split system and the original Jif/split system [61] are both able to successfully partition some program given a trust configuration, then the performance of the two partitioned programs will be the same. However, our system is able to successfully partition a given program for a larger class of trust configurations than the original Jif/split, which does not support replication. In fact, all programs that are compilable by the original Jif/split are compilable in our system; none of the benchmark programs can be compiled by the original Jif/split, as they all require replication.

For example, the players' grids in the Battleship game must be trusted by both principals, but since no host in configuration W is trusted by both principals, the original

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

Jif/split would be unable to find a host on which to store either grid. Our system satisfies the security requirements by replicating the grids on both hosts, though only in hash value form on the opponent's host.

6 Related Work

We have used the term "end-to-end security policies" largely synonymously with "information flow policies". Information flow policies have been enforced using both dynamic [14, 25] and language-based techniques [9, 27, 28, 13, 53, 18, 34, 35, 3, 38]. Jif [29, 31] is a full-scale implementation of a security-typed language. This work builds on the original Jif/split system [61] that introduced the secure partitioning technique, extending it to support automatic replication of code and data.

Although most research on information flow has focused on confidentiality policies, integrity has also been studied [4, 32]. Security types that capture integrity have been used to reason about the correctness of communications protocols [17] and to find format string vulnerabilities in C [43]. Stack inspection [55] also protects integrity by ensuring that privileged code is not invoked by untrusted parties.

Fragment-Redundancy-Scattering (FRS) is a related design methodology in which programmers implement secure applications by manually splitting and replicating their code and data to achieve confidentiality and integrity [51, 12]. Secure program partitioning differs in that programmers write formal security policies into their applications that enable the system to automatically split and replicate code and data in order to enforce a formally specified security condition.

Another language-based approach that uses code transformation to enforce security policies is inline reference monitors [10, 11]. Automated code transformation has also been used to guard against buffer overflows [8] and more generally, violations of memory safety [54]. However, none of these code-wrapping techniques can enforce information-flow policies [41].

Program slicing techniques [56, 50] provide information about the data dependencies in a piece of software. Although the use of backward slices to investigate integrity and related security properties has been proposed [15, 22], the focus of work on program slicing has been debugging and understanding existing software.

7 Conclusions

End-to-end security assurance is a long-standing problem that is growing more important as computation becomes increasingly distributed, spanning organizational and other trust boundaries. Information-flow policies are a natural way to specify end-to-end security, but there has been little prior investigation of how to practically specify and enforce them in systems with mutual distrust and distrusted hosts. Enforcement of data integrity policies is a central problem that was identified in earlier work on secure program partitioning [61] but not satisfactorily resolved.

This paper has described a way of exploiting redundancy to improve integrity guarantees. The definition of security is the same as in the original secure partitioning work: the security policies of a principal can be violated only if a trusted host misbehaves, perhaps because of a successful attack. In this work, we have shown that an extension to secure partitioning in which program code and data are replicated to satisfy the security constraint. The results show that examples of useful secure distributed computation that could not be supported by the original secure partitioning algorithms can be successfully partitioned using replication.

Adding replication involves several nontrivial extensions to the run-time protocols. Because confidentiality and integrity can conflict, data may be replicated onto hosts using a one-way hash that permits integrity verification without violating confidentiality. Untrusted hosts are prevented from sabotaging the integrity of program control flow by a run-time protocol based on capabilities that are decomposed into sets of unforgeable tokens. A synchronization protocol prevents concurrently executing code segments from introducing inconsistencies.

One benefit of programming in a security-typed language is that programmers only need to specify high-level security requirements, and the compiler can generate code that actually fulfills those security requirements. This paper shows that the secure partitioning methodology can be extended to improve support for data integrity. However, there is much left to be done: for example, supporting true concurrent programming, availability policies, and dynamically varying principals and policies.

Acknowledgments

We would like to thank several people who gave useful suggestions on the presentation of this work. In addition to the anonymous reviewers, Lorenzo Alvisi, Kavita Bala, Nate Nystrom, Andrei Sabelfeld, and Stephanie Weirich all helped improve this paper.

References

[1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

[2] Adam Back, Ulf Möller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. *Lecture Notes in Computer Science*, 2137, 2001.

[3] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[5] Manuel Blum. Coin flipping by telephone. In *Advances in Cryptology: A Report on CRYPTO 81*, pages 11–15, 1981.

[6] Gerard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In *Proc. ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395, July 2001.

[7] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (to appear)*, New Orleans, LA, February 1999.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[9] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[10] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigm Workshop*, Caledon Hills, Ontario, Canada, 1999.

[11] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, May 1999.

[12] Jean-Charles Fabre, Yves Deswarte, and Brian Randell. Designing secure and reliable applications using fragmentation-redundancy-scattering: an object-oriented approach. In *PDCS 2: Open Conference*, pages 343–362, Newcastle-upon-Tyne, 1994. Dept of Computing Science, University of Newcastle, NE1 7RU, UK.

[13] Richard J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International Computer Science Lab, Menlo Park, California, January 1980.

[14] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[15] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 154–163, Orlando, FL, 1994. IEEE Computer Society Press.

[16] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[17] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *Preliminary Proceedings of the 17th Conference on the Mathematical Foundations of Programming Semantics (MFPS 17)*, Aarhus, May 2001. BRICS Notes Series NS-01-2, May 2001, pages 99–120.

[18] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[19] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 81–92. ACM Press, January 2002.

[20] Paul A. Karger. Non-discretionary access control for decentralized computing systems. Technical Report MIT/LCS/TR-179, MIT Laboratory for Computer Science, Cambridge, MA, May 1977.

[21] Gregory E. Kersten, Sunil J. Noronha, and Jeffrey Teich. Are all e-commerce negotiations auctions? In *Proc. COOP'2000 : 4th International Conference on the Design of Cooperative Systems,*, Sophia-Antipolis, France, May 2000.

[22] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith. B. Gallagher, Joseph. P. Poole, and David. W. Binkley. Unravel: A CASE tool to assist evaluation of high integrity software. IR 5691, NIST, 1995.

[23] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[24] Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 126–142. IEEE Computer Society Press, June 2001.

[25] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[26] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187, 1990.

[27] Jonathan K. Millen. Security kernel validation in practice. *Comm. of the ACM*, 19(5):243–250, May 1976.

[28] Jonathan K. Millen. Information flow analysis of formal specifications. In *Proc. IEEE Symposium on Security and Privacy*, pages 3–8, April 1981.

[29] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[30] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE

[31] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.

[32] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[33] François Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[34] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.

[35] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[36] Ronald L. Rivest. The MD5 message-digest algorithm. Internet RFC-1321, April 1992.

[37] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer-Verlag.

[38] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[39] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[40] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[41] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[42] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, New York, NY, 1996.

[43] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[44] Geoffrey Smith. A new type system for secure information flow. In *CSFW14*, pages 115–125. IEEE Computer Society Press, June 2001.

[45] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, January 1998.

[46] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. Technical report, Project Athena, MIT, Cambridge, MA, March 1988.

[47] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Operating Systems Design and Implementation (OSDI)*, pages 165–180, San Diego, CA, October 2000.

[48] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[49] David Sutherland. A model of information. In *Proc. 9th National Security Conference*, pages 175–183, Gaithersburg, Md., 1986.

[50] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[51] Gilles Trouessin, Jean-Charles Fabre, and Yves Deswarte. Improvement of data processing security by means of fault tolerance. In *14th National Computer Security Conference*, pages 295–304, Washington, USA, 1991.

[52] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2,3):231–253, November 1999.

[53] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[54] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 203–216. ACM Press, December 1993.

[55] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[56] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[57] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 144–161, May 1990.

[58] Tatu Ylonen. SSH – secure login connections over the Internet. In *The Sixth USENIX Security Symposium Proceedings*, pages 37–42, San Jose, California, 1996.

[59] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001.

[60] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *Proc. 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, 2001.

[61] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)
1081-6011/03 $17.00 © 2003 IEEE