Using Replication and Partitioning to Build Secure Distributed Systems

March, 24 2008

Distributed systems with mutual distrust

- Describe a way to enforce policies for data confidentiality and integrity
- Key: automatic replication of code and data to increase assurance of integrity without harming confidentiality

Challenges

- these systems naturally cross administrative and trust boundaries
- Some of the participants do not trust other participants
- Some of the participants do not trust the computing software and hardware that other participants provide

Solutions

- Secure program partitioning
  - Secure partitioning: security of a principal can be harmed only by the hosts the principal trusts
  - Jif/split compiler partitions high-level, non-distributed code into distributed subprograms
  - Subprograms run securely on a collection of host machines
- Replication
  - Replicated data and computations can be checked against each other to ensure they agree

Replication Challenges

- Trust is heterogeneous
- Replication makes confidentiality policies harder to enforce
- Efficiency

Secure program partitioning

- The desired computation is expressed as a non-distributed program containing annotations
  - These annotations are used to check at compile time
  - The splitter uses these annotations with trust configuration to partition code and data onto hosts
  - Computations that would ordinarily be written as separate programs can be written as a single program
  - Splitter automatically generates separate sub-programs and discovers a network protocol that they may use to communicate

decentralized label model

- defines a set of rules that programs must follow in order to avoid leaks of private information.
- In Jif, these rules can largely be checked statically.
- program annotations (labels) describe the allowed flow of information in a Jif program.
- These labels, in conjunction with ordinary Java type declarations, form an extended type system.
- Jif programs are then type-checked at compile time
  - they are type-safe
  - they do not violate information flow rules.

Security Labels and principals

- In this model Principals can express ownership of information-flow policies
  - A principal is an entity(user,process,…) that can have a security concern.
  - Concerns are expressed as labels
  - Principals can be named as owners of policies and as readers of data
- security label specifying confidentiality is $\{o:r_1,r_2,\ldots,r_n\}$
- security label specifying integrity is $\{*:p_1,p_2,\ldots,p_n\}$
- Labels combining integrity and confidentiality
  - Example: $\{*:p_1;\ p_1:p_2\}$
- Labels on data create restrictions on the use of that data
  - The use of high-confidentiality data is restricted to prevent information leaks
  - The use of low-integrity data is restricted to prevent information corruption

Label relationship

- The relation $\subseteq$ is a pre-order whose equivalence classes form a distributive lattice
- Label relationship: L1 $\subseteq$ L2
  - If L2 specifies at least as much confidentiality as L1, and at most as much integrity as L1
- lattice join and meet operations
  - Join operation combines the restrictions on how data may be used
    - Example: if x has label L1 and y has label L2 then x+y has label L1$\cup$L2
  - Meet operation describes at least as much integrity as either label
    - Example: L1$\cap$L2 is at most as restrictive as L1 or L2
- For any label x, the notations *C*(x) and *I*(x) refer respectively to the confidentiality and integrity parts of x.

Jif(Java Information Flow)

- Contains labels based on the decentralized label model
- Variables and expressions may have security labels type
  - Example: a value with type int{o:r}
  - Labeled type for expression: an upper bound on the security of the data represented by expression
- Jif's type-checking algorithm prevents labeled information from being downgraded, or assigned a less-restrictive label
- Implicit flows can create both integrity and confidentiality concerns
  - Security policies on control flow are expressed as labels
    - Two special labels C(pc) and I(pc) for each program point pc

Jif Programming

- A Jif programmer doesn’t have control over security
  - Labels should be internally consistent
  - Labels must be consistent with security policies in the external environment
- Jif supports two operators for intentionally downgrading security policies: declassify, endorse
  - Declassification: reduces confidentiality requirements
  - Endorsement: increases the claimed integrity of data

```
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

Trust model and security assurance

- H: a set of known hosts, among which the program is to be distributed.
- To partition a program securely, the splitter must know the trust relationships between the participating principals and the hosts H.
- host h has a security label that describes the trust that principals place in h.
- *C*(h) is an upper bound on the confidentiality of information that can be sent securely to h
- *I*(h) is an upper bound on the integrity of information that can be received securely from h
- The *trust configuration* is a map from all the hosts in H to their corresponding security labels

The trust model and security assurance

- A secure partitioning must satisfy the following condition
  - Security assurance: suppose H_{bad} is the set of compromised hosts in the system.
    - The confidentiality of an expression e cannot be harmed unless $C(e) \subseteq \cup\, C(h)$ for $h \in H_{bad}$
    - Its integrity cannot be harmed unless $\cap \mathcal{I}(h) \subseteq \mathcal{I}(e)$ for $h \in H_{bad}$
  - The security assurance condition is not always easy to satisfy
    - Example: label for h_a: {Alice:; *:Alice} and label for h_b: {Bob:; Alice:Bob; *:Bob}

Solution: Partitioning and Replication

- The original secure program partitioning algorithm has been extended to exploit automatic replication
  - Example: extended Jif/split compiler can replicate the field isCommited on both h_a and h_b
- by replicating data on a set of hosts $h_1,\ldots,h_n$, integrity may be increased up to the combined integrity $\mathcal{I}(h_1) \cap \ldots \cap \mathcal{I}(h_n)$ if the replicas all agree

Splitting code, classes and objects

- The splitter partitions a class C into local classes C_1, …, C_n.
- An object o of class C is represented by local objects o_1,…,o_n
  - These local objects share the same *global object ID*
- The code of each secure method is split into *code segments*.
  - A code segment is identified by the source program point *pc* at which the fragment begins
  - Each code segment is replicated on a set of hosts.
- A running method has an activation record, represented as an object in the partitioned target code.
  - Each activation record is partitioned into local frame objects
  - Local frame objects that represent the same activation record share the same *global frame ID.*

Selecting hosts for data

- If a data item d is replicated on hosts $h_1,\ldots,h_n$
  - $C(d) \subseteq C(h_i)$ for all i or equivalently $C(d) \subseteq \cap C(h_i)$
  - $\cap\, I(hi) \subseteq I(d)$
- The hosts holding d may receive access or update request for d
- 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 $C(pc) \subseteq \cap C_{if}(d)$ is satisfied.
  - The host hi must be trusted to read the implicit flow *or* $C_{if}(d) \subseteq \cap C(h_i)$

Selecting hosts for data(summary)

- There is a tension between confidentiality and integrity
  - Solution: 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
  - Hash replicas: hashed copies of a piece of data
  - If host h holds a hash replica of d, it should have a confidentiality label at least as high as $C_{if}(d)$
- For hash replicas, there are three constraints for placing d on hosts $h_1,\ldots,h_n$:
  - $\exists\, i\ C(d) \subseteq \cap C(h_i)$
  - $C_{if}(d) \subseteq C(h_1) \cap \ldots \cap C(h_n)$
  - $\mathcal{I}(h_1) \cap \ldots \cap \mathcal{I}(h_n) \subseteq \mathcal{I}(d)$

example

- bid: {Alice:; *:Alice, Bob}
- C_{if}(bid)= {Alice: Bob}
- Bid is replicated on h_a and h_b
- h_a Can hold its real value
- h_b Can only hold its hash
  - $C(\text{bid}) \subseteq \cap C(h_a)$
  - $C_{if}(d) \subseteq C(h_a) \cap C(h_b)$
  - $I(h_a) \cap I(h_b) \subseteq I(\text{bid})$

```
int{Alice:; *:Alice,Bob} bid;
boolean{Alice:Bob; *:Alice,Bob} isCommitted;

void commit{Alice:Bob; *:Alice,Bob}
       (int{Alice:; *:Alice} v)
         where authority (Bob)
{
  v = (v>=0) ? v : 0;
  if (!isCommitted) {
    bid = endorse(v, {*:Alice,Bob});
    isCommitted = true;
  }
}
int{Alice:Bob; *:Alice,Bob} reveal{*:Alice,Bob} ()
    where authority (Alice)
{
  if (isCommitted)
    return declassify(bid, {Alice:Bob});
  else return -1;
}
```

C_{if}(d) = {Alice:Bob}

Bid is replicated on ha and hb. Ha can hold its real value while hb can only hold its hash value

Selecting hosts for code

- Hosts running a statement s need to read all the inputs of the statement
- $U_r(s)$: set of inputs whose real values are needed in the computation of s
- $U_h(s)$: set of inputs whose hash replicas are sufficient to carry out s
- then $C(s) = \cup C(v) \;\; v \in U_r(s)$
- In general, hosts $h_1,\ldots,h_n$ can execute the statement s securely if the following three constraints are satisfied:
  - $\forall\, v' \in U_h(s)\ \exists\, i\ \ C(v') \subseteq C(h_i)$
  - $C(s) \subseteq C(h_1) \cap \ldots \cap C(h_n)$
  - $\mathcal{I}(h_1) \cap \ldots \cap \mathcal{I}(h_n) \subseteq \mathcal{I}(s)$

Run-time interface

- operations for transferring data between hosts
  - getField
    - To read a field f, a host h sends **getField** request to a host set H_f that hold f
    - Each host in H_f returns the value of f to h after checking $C(f) \subseteq C(h)$
    - h compares the replicas of f from H_f, and accepts it if all replicas are the same
  - setField
    - To update a field f replicated on h'_1 through h'_r, the updating hosts send **setField** request to each h'_i
    - Updating hosts perform update after checking

      $I(h_1) \cap \ldots \cap I(h_n) \subseteq I(a) \cup I(h'_j)$
  - Forward
    - If a code segment updates a local variable, it has to **forward** the update to other code segments residing on remote hosts

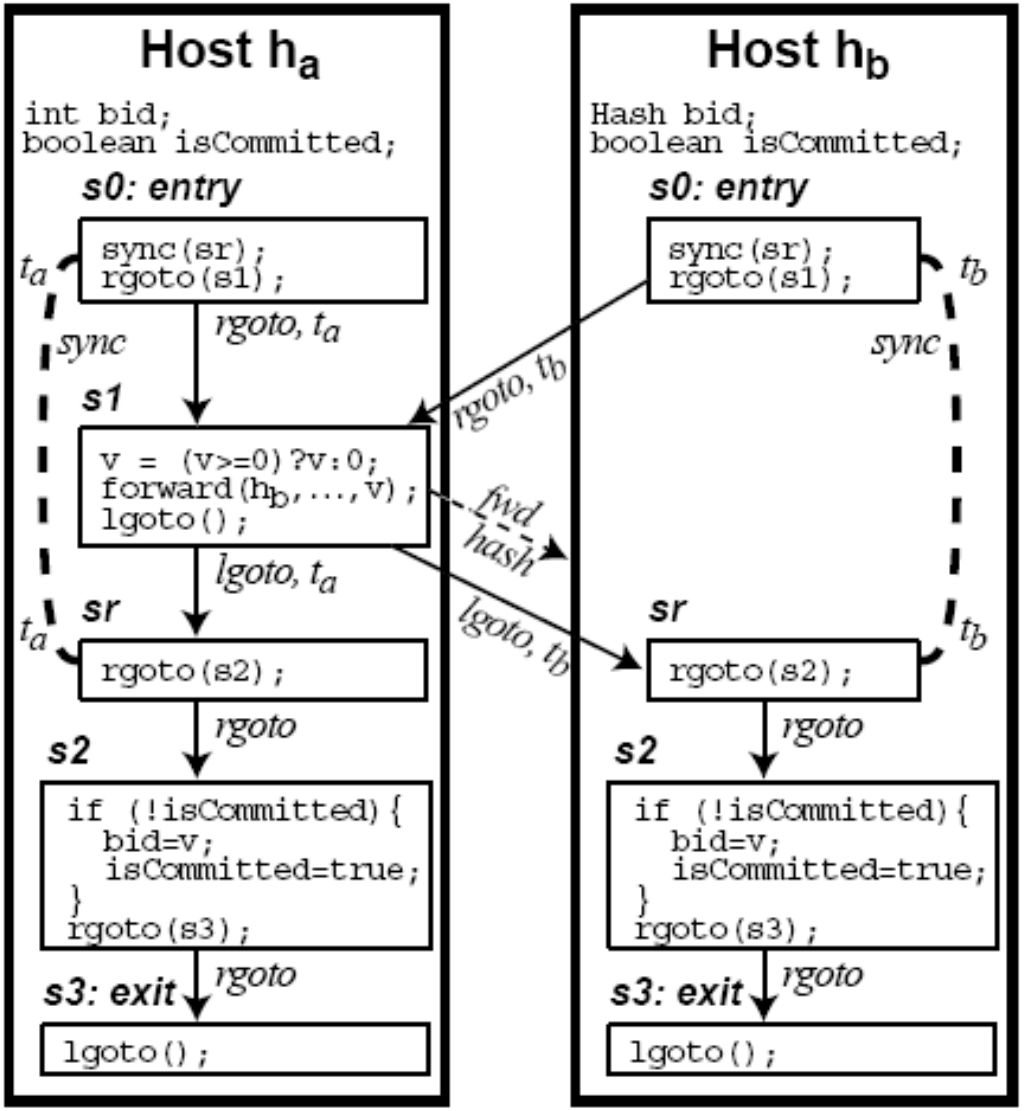
Run-time interface

- operations for transferring control between hosts
  - rgoto transfers control from a code segment to another with equal or lower integrity
  - Lgoto transfers control to another code segment with higher integrity
  - Sync operation is replicated on multiple hosts and creates a set of capability tokens
  - A capability token is a tuple <*h, f, pc, uid*>, containing a host ID, a frame ID, a program counter, and a unique 128-bit identifier.
  - Capability token can be used to invoke the code segment replica on the local host
  - Capability tokens allow low-integrity hosts to invoke high-integrity code segments using lgoto
  - Its critical to restrict the creation and propagation of these tokens

example

- integrity label of s_0: {*:Alice,Bob}
- integrity label of s_1: {*:Alice}
- integrity label of s_2: {*:Alice,Bob}

Control flow graph of the *commit* method



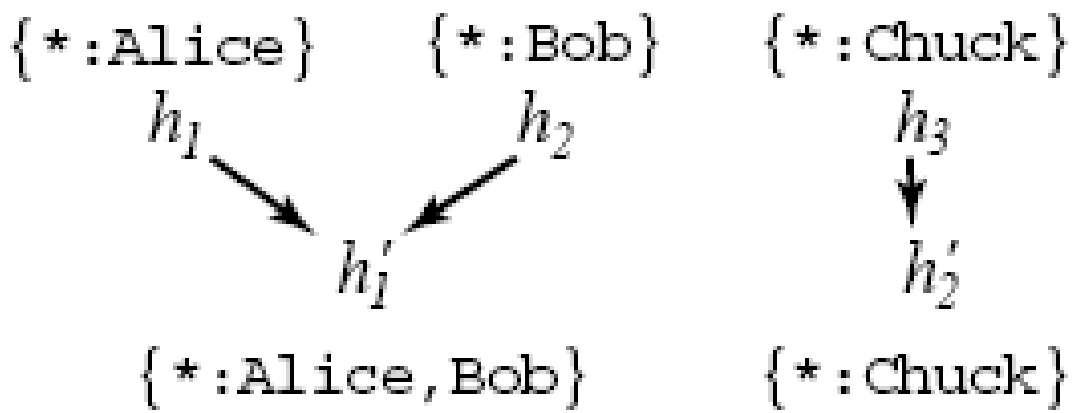
Replication and run-time checks

- a run-time call on host h sends a message m to host h′ to invoke an action a.
- $C(m)$: confidentiality of information in m
- $\mathcal{I}$(a) : integrity required to perform a
- System must enforce two security constraints:
  - $C(\mathrm{m}) \subseteq C(\mathrm{h}')$ (the splitter ensures that when it generates the code for run-time call)
  - $\mathcal{I}(\mathrm{h}) \subseteq \mathcal{I}(\mathrm{a})$ (checked at run time)
- In general, if hosts $\mathrm{h}_1,\ldots,\mathrm{h}_n$ send messages to hosts $\mathrm{h}'_1,\ldots,\mathrm{h}'_r$ to invoke an action a, each h'_j can securely perform a if the following condition holds:
  - $\mathcal{I}(\mathrm{h}_1) \cap \ldots \cap \mathcal{I}(\mathrm{h}_n) \subseteq \mathcal{I}(\mathrm{a}) \cup \mathcal{I}(\mathrm{h}'_j)$

Example: h_1,h_2, and h_3 want to update a field f, replicated on h'_1 and h'_2

Integrity of f: {*:Alice,Bob,Chuck}

$\mathcal{I}(h_1) \cap \ldots \cap \mathcal{I}(h_n) \subseteq \mathcal{I}(a) \cup \mathcal{I}(h'_j)$



conclusions

- Exploiting redundancy to improve integrity guarantees
- Extension to secure partitioning in which program code and data are replicated
- Improvement to support data integrity
- Programmers specify high-level security requirements, the compiler generates code

References

- Zheng, L., Chong, S., Myers, A. C., and Zdancewic, S. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 11 - 14, 2003). SP. IEEE Computer Society, Washington, DC, 236.
- 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.