

Intrusion-Tolerant Dissemination in Large-Scale Systems

Kim P. Kihlstrom, Robin S. Elliott, Kelsey A. Marshman, and Aaryn C. Smith
Computer Science, Westmont College, Santa Barbara, CA, USA

Abstract - *With the growth of the Internet and increased demand for Web Services has come a heightened need for scalable wide-area group communication systems. The need for trust in such systems and the vulnerability of such systems to attack provide motivation to develop protocols that are intrusion-tolerant and able to provide critical services even during an attack or in the presence of arbitrary faults, errors, and accidents.*

We present StarblabIT, a new gossip-based group communication system that is scalable and resilient to process and communication failures, including arbitrary (Byzantine) faults and malicious attack. We employ a modular approach to the design, and build on prior work that generalizes the transformation of crash-tolerant protocols to Byzantine-tolerant protocols. The StarblabIT protocol avoids the need for signatures on most messages by making use of message digests embedded in a single signed message to achieve efficient and secure message delivery.

Keywords: Intrusion-tolerant, gossip protocol, scalable, group communication, secure, Byzantine fault-tolerant

1 Introduction

Reliable group communication that is scalable is important to support a variety of distributed applications and Web Services including distributed databases and publish-subscribe systems. As these types of applications become more and more pervasive and integral to critical services, there is a heightened need for security and reliability in the underlying middleware.

We have developed StarblabIT, a secure, reliable group communication system that is scalable to a large number of nodes distributed over a wide area network. The communication system tolerates a wide range of faults, errors, and failures, including arbitrary and malicious (Byzantine) faults.

In addition to the security and reliability properties provided by our system, our protocol also avoids the use of signatures on every message through a technique in which message digests are embedded in a single signed message and used to validate unsigned messages. Thus, our system achieves good performance for the properties that it provides.

The rest of the paper is organized as follows. We outline related work in Section 2 and describe our protocol in Section 3. We give conclusions and plans for future work in Section 4.

2 Related work

Prior intrusion-tolerant group communication systems that have been designed for a local area network include Ram-part [1], SecureRing [2] and BFT [3]. Intrusion-tolerant systems designed for a wide area network include Phalanx [4], Secure Spread [5], MAFTIA [6], and Worm-IT [7].

There are numerous complexities to be considered in designing an intrusion-tolerant distributed system such as this, *i.e.*, one that must continue to provide useful services even in the event of intrusion, malicious attack, human error, faults, or physical damage that may damage some of the underlying system. To manage the complexity of this task, our approach builds on prior work done in the area of gossip-based communication protocols and in the area of transforming crash-tolerant distributed protocols into intrusion-tolerant protocols.

We have made use of modular transformation techniques as described by Baldoni, *et al.* [8]. Their approach is to specify a design method to take a crash-tolerant, round-based protocol and transform it into an intrusion-tolerant protocol. The transformation of a crash-tolerant group communication system to one that is intrusion-tolerant has also been done by Ramasamy, *et al.* [9].

The crash-tolerant protocol that we began with is a gossip-based communication protocol such as those described by Kermarrec, *et al.* [10]. In such a protocol, each node maintains a partial view of the group membership, to which it forwards (“gossips”) the messages it receives. Our protocol is based on SCAMP [11], a self-organizing group membership protocol. The membership protocol operates in a completely decentralized manner, providing members with a partial view size that is appropriate for the size of the system but without the need for any node to know the group size.

A gossip protocol relies on peer-to-peer communication. Such a protocol is quite scalable to very large systems, and is particularly suitable for systems that are not completely connected, or in which some nodes and links may fail and recover. The load is distributed among the nodes, providing scalability. Pittel [12] showed that, for a gossip-based system, the number of gossip rounds required to reach all the nodes in a system of size n is proportional to $\log(n)$. Resilience to node and link failures is achieved by a redundancy of paths and messages.

The basic communication protocol operates in much the same way as gossip spreads or an infection is disseminated, as follows. When a node p needs to communicate a message m , it sends the message to a randomly chosen set of nodes. When a node q receives m for the first time, it delivers the message and forwards it to a randomly chosen set of nodes. The communication protocol provides probabilistic guarantees that messages will be delivered atomically, *i.e.*, that the messages will reach every node in the system.

Each node in the system maintains both an *inView* of nodes from which it receives gossip messages and an *outView* of nodes to which it forwards messages. The *outView* of each node provides a partial view of the entire system. Whenever a message is received at a node for the first time, that message is then forwarded to all the nodes in the *outView*. The gossip protocol depends on the SCAMP membership protocol to form and maintain the *inView* and *outView* at each node. SCAMP includes mechanisms for a node to join the system, for a node to detect and recover from isolation, and for the underlying graph to be rebalanced. The membership protocol is executed autonomously at each node, without global knowledge of the system.

A new node r joins the system by sending a join message to a node p in the system, called a contact node. The new node r begins with only its contact p in its *outView*. Contact node p then forwards the join message to all nodes in its *outView*. Additionally, p creates extra copies of the join message and forwards each to a random node in its *outView*. The number of extra copies is a design parameter that determines the mean size of the *outViews* in the system.

When a node p receives a forwarded join message originally sent by r , and r is not already in p 's *outView*, p will, with a given probability, keep the join message and add r to p 's *outView*, or else forward the join message to a random node in p 's *outView*. The probability of keeping the join message is inversely related to the size of p 's *outView*, and serves to keep the system balanced in the size of its *outViews* as well as its diameter, both of which are $O(\log(n))$.

Although the join protocol described creates a connected graph, it is possible that link and node failures can cause a node to become isolated. To rectify this situation, each node periodically sends a heartbeat message to the nodes in its *outView*. A node that fails to receive heartbeat messages for a period of time will initiate the process of joining again.

SCAMP makes use of two additional mechanisms to ensure that the graph is balanced, *i.e.*, that the *outViews* are of the required size. These mechanisms are *indirection* and *leases*, and are described in the original protocol [11].

In prior work, we have described Starblab [13], an implementation of the protocol described above. Starblab served as the starting point for StarblabIT, the new intrusion-tolerant system we describe here. Both of these protocols are a part of a larger system, Starfish [14, 15].

3 Intrusion-tolerant protocol

In this section we describe our intrusion-tolerant protocol StarblabIT, beginning with the system model, continuing with the delivery properties, next giving the operation of the protocol, and finally providing implementation details.

3.1 System model

We consider a system consisting of n nodes, where n may be very large, *e.g.*, tens of thousands. None of the nodes needs to be aware of the total number of nodes in the system. The underlying graph formed by the nodes and the links between nodes is assumed to be connected in that every node is reachable by at least one path from every other node, where a path consists of one or more links.

We employ a public key cryptosystem such as RSA [16] in which each node possesses a private key known only to itself with which it can construct a signature $sig(m)$ for a message m . Each node is able to obtain the public keys of other nodes to verify the signature of a message. The protocols also employ a message digest function (also known as a cryptographic hash function) such as SHA-1 [17] in which an arbitrary-length message m is mapped to a fixed-length digest $dig(m)$.

The system is subject to node faults. Nodes are either correct or faulty. *Correct* nodes always behave according to their specification. *Faulty* nodes exhibit arbitrary (*Byzantine*) behavior. We use f to denote the number of faulty nodes. While temporary communication faults (such as the delay or loss of a message) are tolerated, *persistent* communication faults are treated as node faults and must be included in f , since a node that is completely unable to communicate with other nodes cannot be distinguished from a node that has crashed. Correspondingly, the network is assumed not to experience a partition in which more than f correct nodes remain persistently isolated from the others. A node that has behaved in a malicious manner is considered malicious forever. Thus, a malicious node that is subsequently repaired must be handled as a new node.

We model attacks and attackers as follows. A faulty node may attempt to disrupt the system by sending different information to different nodes, purporting that it is the same information, or by selectively sending messages to some nodes and not to others. Further, a malicious node can send messages in a different order to different nodes. Such a node can also masquerade as another node by including the identifier of another node in the messages it sends. A faulty node can refuse to send messages, can send messages that are syntactically incorrect, or can claim that another node is faulty. However, a faulty node is not able to forge the signature of a correct node.

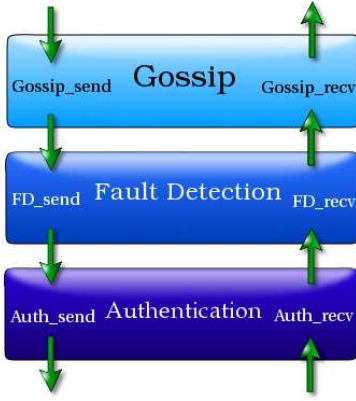


Figure 1. Software architecture.

3.2 Delivery properties

StarblabIT provides *secure delivery* of regular (*i.e.*, application-generated) messages, which is defined in terms of the following properties:

Uniqueness: For any regular message m , every correct node p delivers m at most once.

Authenticity: For any regular message m that contains the identifier of a correct node p , a correct node q delivers m only if m was sent by p .

(Probabilistic) Consistency: If a correct node p delivers a regular message m , then with high probability no correct node q delivers a mutant message m' having the same identifier as m but different contents.

(Probabilistic) Reliability: If p and q are both correct nodes, and p sends a message m , then with high probability q delivers m .

3.3 Protocol operation

To render the crash-tolerant protocol described in Section 2 intrusion-tolerant, we make use of transformation techniques as described by Baldoni, *et al.* [8]. They use a modular approach to transform a crash-tolerant round-based protocol into an intrusion-tolerant one. Their modules include the basic round-based protocol, a muteness failure detector, a non-muteness failure detector, a signature module, and a certification module.

We use this basic approach, although our modules are slightly different. Figure 1 shows the software architecture for StarblabIT, which includes the basic gossip module, a fault detection module, and an authentication module¹. For

¹If confidentiality is desired, an encryption module can be added at the bottom of the stack. The encryption module simply encrypts all messages before sending to the communication medium, and decrypts all messages received from the communication medium. Similarly, if causal or total ordering is desired, an ordering module can be added at the top of the stack.

an application sitting above the gossip module to originate a regular message, it calls the *send* function of the gossip module. The message is passed down the protocol stack and is eventually transmitted over the communication medium by the *send* function of the bottom module. A regular message received from the communication medium at the bottom module is passed up through the protocol stack and is eventually delivered to the application. A received message may also be forwarded by sending it to other nodes.

The gossip protocol handles sending, receiving, forwarding, and delivery of regular messages to the application. The fault detector identifies Byzantine faults. The authentication module signs certain messages, verifies signatures, and performs basic filtering of duplicate and syntactically incorrect messages.

The StarblabIT protocol makes use of several types of messages. The fields² of a message m are:

type: message type; one of {REGULAR, JOIN, JOINACK, OR HEARTBEAT}

sender: identifier of the original sender of m

path: a vector containing the identifiers of nodes that have forwarded m

seq: regular sequence number for a REGULAR message, join sequence number for a JOIN or JOINACK message, or heartbeat sequence number for a HEARTBEAT message

payload: application-generated contents of a REGULAR message, list of digests of previous regular messages for a HEARTBEAT message, sender's public key for a JOINACK message, or \perp for a JOIN message

authen: a digital signature for a HEARTBEAT message, or \perp for all other types

Each node p maintains a buffer $sent_p$ of messages it has sent and a buffer $recv_p$ of messages it has received. The $sent_p$ buffer contains, for each entry, the following information:

m : the message

referred: a boolean indicating if p has sent a HEARTBEAT message referring to this message, as described in Section 3.3.2

The buffer $recv_p$ contains, for each entry, the following information:

m : the message

referrals: a list of senders who sent HEARTBEAT messages referring to this message, as described in Section 3.3.2

²Throughout this paper, we use \perp to refer to an empty value and \emptyset to refer to the empty set. When an empty message is created, all of the fields contain either \perp or \emptyset as appropriate.

mutant: a boolean indicating if the message is a mutant of another message

deliverable: a boolean indicating if the message has satisfied the criteria for delivery to the application, as described in Section 3.3.2

There are two system-wide design parameters *extra* and *f* that must be supplied upon initialization: *extra* determines the mean size of the *outViews* and *f* is the number of faults tolerated. In addition, each node must be supplied with a local list *contacts_p* of at least $f+1$ contact nodes. All local values (e.g., *contacts_p*, *recv_p*, *sent_p*) are accessible from any module at *p*. Pseudocode for initialization is given in Figure 2.

```

Main_init()
{
  recv_p ← ∅;
  sent_p ← ∅;
  extra_p ← extra;
  Gossip_init();
}

```

Figure 2. New node *p* starts, given design parameters *extra*, *f* and a list *contacts_p* containing identifiers of $f+1$ nodes.

```

Gossip_init()
{
  seq_p ← 0;
  joinSeq_p ← 0;
  inView_p ← ∅;
  FD_init();
  create empty message j;
  j.type ← JOIN;
  j.sender ← p;
  joinSeq_p++;
  j.seq ← joinSeq_p;
  choose randomly node q ∈ contacts_p;
  FD_send(j,q);
}

```

Figure 3. New node *p* starts gossip protocol.

```

Gossip_send(data content)
{
  create empty message m;
  m.type ← REGULAR;
  m.sender ← p;
  seq_p++;
  m.seq ← seq_p;
  m.payload ← content;
  for all nodes q ∈ outView_p do
    FD_send(m,q);
}

```

Figure 4. Gossip module at node *p* sends a regular message, given application-specific data *content*.

3.3.1 Gossip module. The gossip module at a node maintains a set *outView_p* of nodes to which it sends gossip messages, a set *inView_p* of nodes from which it receives gos-

```

Gossip_rcv(message m)
{
  if m ∉ sent_p then
    for all nodes q ∈ outView_p do
      FD_send(m,q);
  if (m marked deliverable in recv_p) then
    Deliver(m);
}

```

Figure 5. Gossip module at node *p* receives a regular message *m*.

```

Gossip_rcv(message j)
{
  if (j.sender ∉ faulty_p) then
    for all nodes q ∈ outView_p do
      FD_send(j,q);
  for (i = 0; i < extra_p; i++) do
    choose randomly q ∈ outView_p;
    FD_send(j,q);
}

```

Figure 6. Gossip module at contact node *p* receives a join message *j* directly from joining node.

sip messages, a sequence number *seq_p* that provides source-ordering of regular messages, a sequence number *joinSeq_p* that provides source-ordering of join messages, and a value *extra_p* that is a design parameter described below. Pseudocode for the gossip module is given in Figures 3 to 7. When a node starts up, it sends a JOIN message to a node randomly chosen from the nodes in its contact list. Rejoining, indirection, and lease mechanisms remain the same as in the original SCAMP [11] protocol. When a message meets the delivery criteria as described in Section 3.3.2, it is delivered to the application.

3.3.2 Fault detector module. The fault detector module builds on prior work in the area of Byzantine fault detection [18]. The fault detector module at a node maintains a sequence number *hbSeq_p* that provides source-ordering of heartbeat messages, a set *faulty_p* of nodes it knows to be faulty, a set *suspected_p* of nodes it suspects based on timeouts, and a set *expected_p* of messages for which a timeout has expired. The fault detector sets timeouts for messages that are expected and regularly checks to see if timeouts have expired. Pseudocode for the fault detector module is given in Figures 8 to 15.

When the gossip module sends a JOIN message, the fault detector module then sets a timeout for receiving JOINACKS from $f+1$ nodes who have kept a forwarded copy of the JOIN message. If the timeout expires, it re-sends the JOIN message to another randomly chosen node. It repeats this process until it has received responses indicating that other nodes have placed it in their *outView*. This mechanism ensures that a faulty node cannot prevent a new node from joining.

The fault detector module performs periodic sending of heartbeat messages. In the original gossip protocol described in Section 2, heartbeat messages are sent periodically by a

```

Gossip_rcv(message j)
{
  if (j.sender  $\notin$  faultyp) then
    if (j.sender  $\notin$  outViewp and with probability  $\frac{1}{1+|outView_p|}$ )
      then
        outViewp  $\leftarrow$  outViewp  $\cup$  j.sender;
        create empty message a;
        a.type  $\leftarrow$  JOINACK;
        a.sender  $\leftarrow$  p;
        a.seq  $\leftarrow$  j.seq;
        a.payload  $\leftarrow$  public key of p;
        FD_send(a, j.sender);
      else
        choose randomly q  $\in$  outViewp;
        FD_send(j, q);
}

```

Figure 7. Gossip module at node p receives a join message j that has been forwarded by a contact node.

```

FD_init()
{
  faultyp  $\leftarrow$   $\emptyset$ ;
  suspectedp  $\leftarrow$   $\emptyset$ ;
  expectedp  $\leftarrow$   $\emptyset$ ;
  hbSeqp  $\leftarrow$  0;
}

```

Figure 8. New node p starts fault detector protocol.

```

FD_timeout(message m)
{
  expectedp  $\leftarrow$  expectedp  $\cup$  m;
  suspectedp  $\leftarrow$  suspectedp  $\cup$  m.sender;
  inViewp  $\leftarrow$  inViewp  $-$  m.sender;
}

```

Figure 9. A timeout expires at node p for a regular or heartbeat message m .

```

FD_send(message j, node q)
{
  Authen_send(j, q);
  set a timeout for receiving  $f + 1$  JOINACK messages;
}

```

Figure 10. Fault detector module at joining node p sends a join message j to contact node q .

```

FD_timeout(message j)
{
  choose randomly node q  $\in$  contactsp;
  find prior JOIN message j in sentp;
  FD_send(j, q);
  set a timeout for receiving  $f + 1$  JOINACK messages;
}

```

Figure 11. A timeout expires at joining node p for $f + 1$ joinAck messages j .

```

FD_rcv(message j)
{
  if ( $f + 1$  JOINACK messages have been received)
    cancel timeout for receiving  $f + 1$  JOINACK messages;
  Gossip_rcv(j);
}

```

Figure 12. Fault detector module at joining node p receives a joinAck message j .

```

FD_rcv(message m)
{
  if (m is a mutant of another message m') then
    Mark m and m' in rcvp as mutants;
    Find heartbeat msgs h, h' in rcvp: dig(m)  $\in$  h.payload and
    dig(m')  $\in$  h'.payload and use m.path and m'.path
    to determine faulty node r;
    faultyp  $\leftarrow$  faultyp  $\cup$  r;
    inViewp  $\leftarrow$  inViewp  $-$  r;
    outViewp  $\leftarrow$  outViewp  $-$  r;
  else
    Cancel any timeout that has been set for receiving m;
    expectedp  $\leftarrow$  expectedp  $-$  m;
    if (no messages in expectedp from m.sender) then
      suspectedp  $\leftarrow$  suspectedp  $-$  m.sender;
    Set referrals for m in rcvp to  $\emptyset$ ;
    for all heartbeat msgs h in rcvp: dig(m)  $\in$  h.payload do
      if h.sender  $\in$  inViewp then
        Add h.sender to referrals for m in rcvp;
    if (referrals for m in rcvp from all nodes r  $\in$  inViewp) then
      mark m in rcvp as deliverable;
    Gossip_rcv(m);
}

```

Figure 13. Fault detector module at node p receives a regular message m .

node, and if a node fails to receive any heartbeat messages for a given time period it concludes that it has become isolated and rejoins. In the intrusion-tolerant protocol, heartbeat messages are used, in addition, to detect mutant messages and to establish criteria for secure delivery of a regular message as described below. Unlike in the original protocol, received heartbeat messages are gossiped to the *outView*.

When a node sends a heartbeat message (as it does periodically), it includes in the message payload a list of message digests for messages it has either originated or forwarded since sending its most recent prior heartbeat message. As described in Section 3.3.3, the node then digitally signs the heartbeat message. These techniques are similar to those used in prior work [2], and provide a way of detecting mutant messages as well as providing the basis for the delivery criteria.

As in prior work [2], we define the “refers to” relation \succ for regular messages and heartbeat messages as follows. If m is a regular message originated or forwarded by a node p , h is the subsequent heartbeat sent by p , and h contains the digest of m in the *payload* field, then $h \succ m$. A node can deliver a regular message m when it has received, from every node in its *inView*, a heartbeat message h such that $h \succ m$. Although space prohibits a formal proof, we briefly sketch the argument for correctness here.

For a graph of the type we consider, the probability that a path exists from a node to all of the other nodes depends only on the mean size of the *outViews*. The mean size k of the *outViews* is determined by the design parameter $extra_p$ as follows:

$$k = (extra_p + 1) \log(n) \quad (1)$$

where n is the number of nodes in the system. As shown in prior work [10], there is a very high probability that atomic

```

FD_rcv(message h)
{
  if (h is a mutant of another message h') then
    Mark h and h' in rcvp as mutants;
    faultyp ← faultyp ∪ h.sender;
    inViewp ← inViewp - h.sender;
    outViewp ← outViewp - h.sender;
  else
    Cancel any timeout that has been set for receiving h;
    Set timeout for receiving next heartbeat from h.sender;
    expectedp ← expectedp - h;
    if (no messages in expectedp from h.sender) then
      suspectedp ← suspectedp - h.sender;
    for all nodes q ∈ outViewp do
      FD_send(h, q);
    for all regular msgs m : dig(m) ∈ h.payload do
      if m ∉ rcvp then
        Set a timeout for receiving m from h.sender;
      else
        Add h.sender to referrals for m in rcvp;
        if (referrals for m in from all nodes r ∈ inViewp) then
          mark m in rcvp as deliverable;
          Gossip_rcv(m);
}

```

Figure 14. Fault detector module at node p receives a heartbeat message h .

broadcasts will be achieved in a system such as this, even in the face of node and link failures.

Consider the case of a Byzantine faulty node p that sends mutant messages m and m' to disjoint subsets of nodes. Since every correct node that receives m will forward it to all of the nodes in its *outView*, then with high probability every node will receive m . Similarly, since every correct node that receives m' will forward it to all of the nodes in its *outView*, then with high probability every node will receive m' . Thus, with high probability the mutant messages will be detected by every node.

To deliver a message m , a node p must have received, from all nodes in its *inView*, signed heartbeat messages that refer to m . Once p has received this set of heartbeat messages, it can with high probability deliver the message as unique and not a mutant, since nodes in its *inView* are the only ones from whom it receives messages, and none of the correct nodes in its *inView* will send a heartbeat message referring to m' as well as to m . The node can afford to wait for heartbeat messages from all of these nodes due to the properties of the fault detector, which ensure that nodes that have failed will eventually be detected and thus removed from the *inView*.

3.3.3 Authentication module. The authentication module filters out duplicate messages that are received, as well as messages that are not properly formed, *i.e.*, that are syntactically incorrect. Pseudocode for the authentication module is given in Figures 16 to 17.

The authentication module handles signing a heartbeat message when it is sent and verifying a signature on a heartbeat message that is received.

```

FD_heartbeat()
{
  create empty message h;
  h.type ← HEARTBEAT;
  h.sender ← p;
  hbSeqp++;
  h.seq ← hbSeqp;
  for all regular messages m ∈ sentp not marked referred do
    h.payload ← h.payload ∪ dig(m);
    mark m in sentp as referred;
  for all nodes q ∈ outViewp do
    FD_send(h, q);
}

```

Figure 15. Fault detector module at node p sends a heartbeat message m (executed periodically).

3.4 Implementation Details

The implementation of the Starblab protocol consists of approximately 4,000 lines of C++ code. Our implementation is available for download from our website [19]. It is organized into modules, which include the main entry module as well as node, contact, and message classes.

```

Auth_send(message m, node q)
{
  if (m.type = HEARTBEAT) then
    m.authen ← sig(m);
    sentp ← sentp ∪ m;
    send(m, q);
}

```

Figure 16. Authentication module at node p sends a message m of any type to node q .

```

Auth_rcv(message m)
{
  if (m ∉ rcvp) then
    if (m is not properly formed) then
      return;
    else if (m.type = HEARTBEAT and m.authen not valid) then
      return;
    else
      rcvp ← rcvp ∪ m;
      FD_rcv(m);
}

```

Figure 17. Authentication module at node p receives a message m of any type.

The node class implements the main event loop and is multi-threaded. A node contains two sockets: an internet datagram socket for communication with other nodes, and a unix stream socket for internal communication with the applications residing on the local machine. A listener thread listens for messages on the sockets.

Execution begins in the main module and an instance of the node class is created. The node constructor initializes the sockets and threads. Control then passes to the main event loop. The node processes messages that are received and sends messages according to the protocol specifications.

4 Conclusions and future work

We have presented StarblabIT, a new intrusion-tolerant communication system. The protocol is quite scalable to a large number of nodes and yet provides a high level of secure and reliable delivery properties. Our system is able to operate efficiently by avoiding the need for signatures on most messages as well as the need for extra rounds of message exchange in normal operation. To achieve this, we make use of a technique in which a single signature can authenticate multiple messages.

The StarblabIT protocol also serves to validate prior work on transforming a crash fault-tolerant protocol to one that is able to withstand malicious attack, corruption, and arbitrary (Byzantine) faults. Demonstrating that these techniques are applicable to a wide range of protocols serves the general community by adding to the toolkits available to systems designers working in the area of intrusion-tolerant systems.

We have implemented StarblabIT and are currently conducting performance testing. Future work will include system tuning, an evaluation of performance tradeoffs, and a comparison of the performance of the original crash-tolerant Starblab system with the new intrusion-tolerant StarblabIT system.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0534167.

References

- [1] M. K. Reiter, “The Rampart toolkit for building high-integrity services,” in *Theory and Practice in Distributed Systems. Lecture Notes in Computer Science 938*. Springer-Verlag, 1995, pp. 99–110.
- [2] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, “The SecureRing group communication system,” *ACM Transactions on Information and System Security*, vol. 4, no. 4, pp. 371–406, Nov. 2001.
- [3] M. Castro and B. Liskov, “Practical Byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
- [4] D. Malkhi and M. Reiter, “Secure and scalable replication in Phalanx,” in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, Oct. 1998, pp. 51–58.
- [5] Y. Amir, C. Nita-Rotaru, J. Stanton, and G. Tzudik, “Secure Spread: An integrated architecture for secure group communication,” *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 3, pp. 248–261, July–September 2005.
- [6] P. Veríssimo, N. F. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch, “Intrusion-tolerant middleware: The road to automatic security,” *IEEE Security and Privacy*, vol. 4, no. 4, pp. 54–62, July/August 2006.
- [7] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo, “Worm-IT - a wormhole-based intrusion-tolerant group communication system,” *Journal of Systems and Software*, vol. 80, no. 2, pp. 178–197, Feb. 2007.
- [8] R. Baldoni, J.-M. Helary, and M. Raynal, “From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach,” in *Proceedings of the International Conference on Dependable Systems and Networks*, New York, NY, USA, Jun. 2000, pp. 273–282.
- [9] H. V. Ramasamy, P. Pandey, M. Cukier, and W. H. Sanders, “Experiences with building an intrusion-tolerant group communication system,” *Software: Practice and Experience*, vol. 38, no. 6, pp. 639–666, May 2008.
- [10] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh, “Probabilistic reliable dissemination in large-scale systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 248–258, 2003.
- [11] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, “Peer-to-peer membership management for gossip-based protocols,” *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 139–149, 2003.
- [12] B. Pittel, “On spreading a rumour,” *SIAM Journal on Applied Mathematics*, vol. 47, no. 1, pp. 213–223, Feb. 1987.
- [13] K. P. Kihlstrom, J. L. Stewart, N. T. Lounsbury, A. J. Rogers, and M. C. Magnuson, “Implementation and performance testing of a gossip-based communication system,” in *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, Nov. 2007, pp. 194–199.
- [14] K. P. Kihlstrom, P. Narasimhan, C. Phillips, C. Ritchey, and B. LaBarbera, “The architecture of the Starfish system: Mapping the survivability space,” in *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, Nov. 2003, pp. 833–843.
- [15] K. P. Kihlstrom and P. Narasimhan, “The Starfish system: Providing intrusion detection and intrusion tolerance for middleware systems,” in *Proceedings of the IEEE Workshop on Object-oriented Real-time Dependable Systems*, Guadalajara, Mexico, Jan. 2003, pp. 191–199.
- [16] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
- [17] National Institute of Standards and Technology, “Secure hash standard,” *Federal Information Processing Standards Publication 180-1*, Apr. 1995.
- [18] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, “Byzantine fault detectors for solving consensus,” *The Computer Journal*, vol. 46, no. 1, pp. 16–35, 2003.
- [19] Westmont College, *The Starfish Project Website*, <https://starfish.westmont.edu/>.