FIRE: Flexible Intra-AS Routing Environment

Craig Partridge, Alex C. Snoeren†, W. Timothy Strayer,
Beverly Schwartz, Matthew Condell, and Isidro Castiñeyra

BBN Technologies
10 Moulton Street, Cambridge, MA 02138

{craig, snoeren, strayer, bschwart, condell, isidro}@bbn.com

ABSTRACT

Current routing protocols are monolithic, specifying the algorithm used to construct forwarding tables, the metric used by the algorithm (generally some form of hop-count), and the protocol used to distribute these metrics as an integrated package. The Flexible Intra-AS Routing Environment (FIRE) is a link-state, intra-domain routing protocol that decouples these components. FIRE supports run-time-programmable algorithms and metrics over a secure link-state distribution protocol. By allowing the network operator to dynamically reprogram both the information being advertised and the routing algorithm used to construct forwarding tables in Java, FIRE enables the development and deployment of novel routing algorithms without the need for a new protocol to distribute state. FIRE supports multiple concurrent routing algorithms and metrics, each constructing separate forwarding tables. By using operator-specified packet filters, separate classes of traffic are routed using completely different routing algorithms, all supported by a single routing protocol.

This paper presents an overview of FIRE, focusing particularly on FIRE's novel aspects with respect to traditional routing protocols. We also briefly describe our implementation experience.

1. INTRODUCTION

A routing protocol has three constituent functions: it defines a set of metrics upon which routing decisions are made; it distributes this information throughout the network; and it defines the algorithms that determine the paths that packets use to traverse the network. Furthermore, a well-designed protocol contains security mechanisms to protect the routing infrastructure from attack as well as from mischance or misconfiguration.

† Alex C. Snoeren is with the MIT Laboratory for Computer Science (snoeren@lcs.mit.edu). This paper describes work carried out while at BBN Technologies.

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract No. DABT63-96-C-0100. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied.

SIGCOMM'00, Stockholm, Sweden.

In today's routing protocols, these functions are tightly integrated and cannot be unbundled. When a network operator chooses to use IS-IS [8] or OSPF [25], for instance, the information that is distributed about each link and the algorithm that is used to select paths are fixed; the operator is largely unable to tune the system to use a new algorithm or different metrics. The operator may select a more recent routing algorithm by moving to Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP) [37], but the operator is then forced to accept EIGRP's metrics, state distribution and security mechanisms.

FIRE, the Flexible Intra-AS Routing Environment, is an attempt to provide a more flexible routing system. Operators control a variety of key routing functions, including choosing which algorithms are used to select paths, choosing what information is used by the algorithms, and identifying traffic classes to be forwarded according to the specified algorithms. Expressing these ideas a bit more formally, FIRE splits the standard routing protocol into its constituent parts: secure state distribution, computation of forwarding table(s), and the generation of state information (i.e., determining what values to distribute). FIRE then exposes its state distribution functionality, making computation of forwarding tables and the generation of state information programmable at run-time.

The motivation for the novel aspects of FIRE springs directly from three simple observations:

- Routing algorithms continue to evolve.
- Today's crude link metrics are often insufficient to support new routing algorithms.
- Network providers are increasingly interested in providing specialized routing for different classes of traffic.

Over the past several years there has been considerable ferment and change in the design of multicast routing protocols [6, 14, 17]. Potential improvements for unicast routing have also been developed [4, 34]. Getting any of these algorithms actually deployed is difficult. Because current routing protocols have intertwined their algorithm with their state distribution mechanism, deploying new algorithms has meant, in most cases, implementing entirely new protocols. In short, the barrier to deploying new routing algorithms is very high. Indeed, part of the motivation for FIRE came from our experience trying to deploy a new unicast routing algorithm that finds approximately optimal paths based on multiple, orthogonal link metrics [9].

Most protocols use hop count to approximate the cost of a link. More sophisticated protocols like EIGRP compute the metric from a mix of several link properties, but the EIGRP equation for combining metrics is fixed and represents a balance between possibly conflicting values. FIRE distributes a rich set of properties for use by routing algorithms. Properties can be pre-configured, extracted from router MIBs, or even dynamically generated by operator-provided applets. FIRE provides for programmable property generation, and properties can be regenerated when conditions suggest forwarding tables need updating.

The increasing heterogeneity of Internet connectivity strongly suggests that there is a growing diversity in path choices. Different paths between two points will be better suited for different applications. An obvious example is satellite links, which are being used more frequently in the Internet and which offer high bandwidth but also high delay. Most routing protocols in use today forward all traffic using the same forwarding table. Traffic classes may be differentiated with respect to resource reservation[7] and queuing priority [28], but, regardless of priority, packets are still routed along the same path. In FIRE, each packet is routed along a path using a forwarding table constructed to best suit its particular traffic class. By assigning different classes of traffic to separate forwarding tables, FIRE allows network operators to optimize routing for each traffic class. Each forwarding table is constructed with a different algorithm, which may use its own set of metrics. Note that class assignment is facilitated by a set of packet filters specified on a domain-wide basis, in contrast to active network techniques that require packets themselves to explicitly specify their routing algorithm of choice, as in PLANet [16].

This paper is organized as follows. Section 2 provides a brief overview of FIRE, while sections 3, 4, 5, and 7 present novel aspects of FIRE that are particularly interesting when compared to existing protocols, including configuration and management, programmable routing algorithms, dynamic properties, and our security model. Section 6 describes FIRE's state distribution mechanism in more detail. Section 8 provides a quick tour of the programming interface to FIRE, and section 9 very briefly discusses our implementation. Section 10 surveys related work, while section 11 concludes with a discussion of future work.

2. FIRE OVERVIEW

A traditional routing protocol generates a single forwarding table at each router, which the router then uses to forward incoming traffic. FIRE extends that notion by generating a set of forwarding tables, each uniquely defined by three pieces of information: the algorithm used to compute the table, the properties used by the algorithm in its computations, and a packet filter that determines which classes of traffic use the forwarding table. In FIRE, all three of these variables may be configured by the network operator at run time.

A single instance of FIRE may manage several forwarding tables, thus serving several classes of traffic, concurrently. Traffic is classified using operator-specified filters and forwarded according to generated forwarding tables. The tables are generated using routing algorithms that are downloaded and installed by the operator on demand. The algorithms are run on a link-state database, called the *property repository*, containing properties that are continually flooded throughout the network through the use of State Advertisements (SAs). These properties may be statically configured, obtained from routers' Management Information Bases (MIBs), or dynamically generated.

FIRE also permits multiple instances of the FIRE protocol to be running concurrently in a system. Each FIRE instance is wholly self-contained, propagates its own state, and maintains a separate set of forwarding tables. This feature is designed to support Virtual Private Networks (VPNs), where overlays are used to make a single network look like several independent networks. A network provider can assign a distinct instance of FIRE to each VPN, and allow the operators to choose how to route different classes of traffic within their own private network.

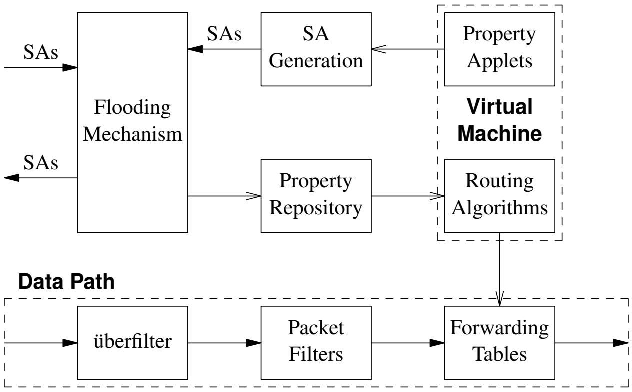

Figure 1: FIRE Router Architecture

The internal architecture of a FIRE router is shown in figure 1. All incoming traffic on the router's data path first passes through an *überfilter* that assigns the traffic to a particular instance of FIRE. A packet can belong to only one instance, hence the filters must be disjoint. (In a router with only one instance of FIRE, the überfilter would contain a single wildcard entry.) Within a FIRE instance, packet filters determine which forwarding table the packet is to use. The packet's destination address is looked up, and the packet is forwarded accordingly.

Routing information is processed by FIRE's flooding mechanism, which ensures SAs generated by any entity in the network are seen by all others. Properties contained in these SAs are stored in a property repository, which is used by the routing algorithms to generate their respective forwarding tables. Each router is also responsible for advertising its own properties and periodically floods its SAs into the network.

2.1. Design Philosophy

The most basic contribution of FIRE is the ability to modify the routing algorithms and property metrics used to generate forwarding tables. This dynamism requires great care to ensure robust, reliable behavior; the sheer scope of configuration and, even more importantly, programming options made available by FIRE's model substantially increases the chance of misconfiguration or buggy implementations. This vulnerability has guided our design of the FIRE protocol. We have sought to make FIRE as stable a platform as possible.

FIRE alleviates many of the security and performance constraints of traditional active networking by providing a strict separation between control and data flow. Unlike previous active network approaches for extensible routing [16, 36], the routing environment is completely separate from the traffic to be routed. The decoupled nature of FIRE's property applets and routing algorithms allows FIRE to achieve forwarding performance similar to traditional routing protocols while providing the rapid extensibility and flexibility of active networking. Further, it greatly limits the scope of additional robustness concerns, since no new threats are posed by the traffic itself.

One vital design decision was to make FIRE a link-state routing protocol. While there remains considerable debate about relative strengths and weaknesses of link state and distance vector, from the FIRE perspective link state has two advantages. First, there are well-defined mechanisms to make the distribution of link-state information robust against Byzantine failures [29] while, to our knowledge, no such mechanisms exist for distance vector.

Hence link state is more secure against certain failures of individual routers. Second, the incremental updating of distance-vector information at each hop in a path means the correct routing of packets in a distance vector environment potentially depends on the correct operation of all routers in the network, while link state only requires correct operation along the desired path. Again, link state appears more robust.

The desire for robustness can also be seen in FIRE's security model, which is based on the philosophy of containment. SAs are cryptographically signed by their creator to prevent modification in flight; advertisements can be suppressed or damaged in flight but cannot be surreptitiously modified or spoofed. Furthermore, each FIRE entity has an associated authorization certificate specifying what information it is allowed to advertise. These certificates are used, for example, to prevent a malicious or misconfigured router from advertising a route to a distant portion of the network and ''black-holing'' traffic.[1]

An additional layer of protection is provided by sending all FIRE traffic over a special transport protocol, the FIRE Layer InterNetwork Transport Protocol (FLINT). FLINT is designed to facilitate the use of IPsec [19] to authenticate hop-by-hop traffic. In addition, FIRE uses an internal hard-coded algorithm to build a special forwarding table for FLINT traffic that cannot be redefined. Hence FLINT traffic (and thus FIRE control and management traffic) will continue to be correctly forwarded even if bugs in a deployed routing algorithm cause all other traffic to be misdirected.

2.2. Operator Configuration

Central to the FIRE model is the notion of an *Operator*. FIRE works within a particular Autonomous System (AS)—an area completely controlled by one administrative entity. That entity appoints one or more people (e.g., a network operations center personell) as the Operator. The Operator is authorized to configure the network through two mechanisms: the Operator Configuration Message (OCM) and the Operator Configuration File (OCF).

The OCM is a special SA message that contains a few system wide configuration values, a list of the OCFs to load, and instructions about which OCFs to run.

An OCF lists the algorithms used to compute forwarding tables, the properties each entity must advertise, and a set of filters used to map an incoming packet onto the proper forwarding table. OCFs are stored on file repositories throughout the network to provide redundant availability in the face of network partitioning; the location of these file repositories are specified in the OCM. When the Operator issues an OCM containing a new OCF, each node retrieves all pertinent files specified by the OCF.

2.3. Algorithms

Routing algorithms in FIRE are downloaded Java programs, available from file repositories as directed by the OCF. The algorithms are designed to use distributed network properties from the property repository to generate a local forwarding table. Each instance of an algorithm (multiple forwarding tables may be generated using the same algorithm with different property metrics) is run in a separate Java Virtual Machine (JVM) on the router itself. This *sandboxing* prevents a buggy or malicious routing algorithm from disabling the entire router.

[1] The term *black-hole* comes from a particular router failure mode in which the router advertises that it is zero hops from from one or more prefixes that it actually does not serve. The result is traffic for those prefixes gets sent to that router, never to emerge again.

2.4. State Distribution

A FIRE system is composed of two classes of entities: nodes and links. Routers and broadcast networks are nodes; links are unidirectional adjacencies between nodes. Each entity in the FIRE system has a unique ID and a set of properties associated with its ID. The ID encoding specifies whether the entity is a link, subnet, or router.

A node is responsible for generating values for all of the properties listed in the OCF. Property values for links are generated by adjacent nodes. Network properties are the responsibility of Designated Routers, which are selected using a distributed election process similar to OSPF. Some of these values may be configured into the nodes (e.g., multicast support or policy-based cost values). Others may be readily available from the router's MIB (e.g., average queue length and CPU utilization). FIRE also allows operators to write their own property applets. Like algorithms, property applets are written Java and each is executed in its own JVM instance.

In addition to internal properties communicated throughout the FIRE domain, routers serving as gateways to other ASs may need to communicate with BGP or a similar Inter-AS protocol. Even within an AS, there may be a need to support concurrent routing protocols for other purposes, such as multicast, if they have not yet been implemented in FIRE. In the case of an external routing protocol such as BGP, the Inter-AS protocol needs to inject summaries of external routes and summarize routes of internal networks. FIRE uses special External Route advertisements to support both classes of information exchange.

All property values are distributed to every node in the network using reliable flooding. Each node stores these values in a property repository in order to build a complete and consistent map of the network. Updates of the values of these properties are periodically flooded throughout the network, refreshing the repositories.

3. CONFIGURATION & MANAGEMENT

Control and management of a FIRE system rests with the Operator. Through the use of an Operator Control Message (OCM), the Operator can specify the Operator Configuration Files (OCFs) to load.

3.1. Configuration Messages and Files

The OCM itself is rather small and contains the configuration rules for the FIRE system, the set of OCFs that are to be loaded, and names one of them as the *running* OCF. The OCM is cryptographically signed with the Operator's secret key. There can be only one OCM valid at any point in time. The OCM is injected into the network by an Operator at any FIRE node, and is distributed along with normal routing updates throughout the network by the standard flooding mechanism.

Upon receipt of the OCM, a node retrieves the listed OCFs from one of the file repositories specified in the OCM. File retrieval is facilitated by a special, simple file transfer protocol called the Large Data Transfer Protocol (LDTP). This protocol is based on the Trivial File Transfer Protocol (TFTP) [33], altered to run over FLINT and enhanced to protect against insertion attacks and reduce its vulnerability to Denial-of-Service attacks. When an OCF is retrieved from a file repository, the OCF is parsed and any additional support files downloaded. In particular, the OCF contains the a list of routing algorithms and property applets, along with a list of the file repositories where these files can be obtained. LDTP is again used to retrieve these files. All files to be downloaded are cryptographically signed to ensure integrity.

The OCM tags each OCF with one of three directives: *load, advertise*, or *run*. An OCF load directive simply causes the router to retrieve the files from the file repository. An OCF *advertise* directive additionally forces the generation of property values and their issuance as SAs once the OCF has finished loading. In addition to the steps required by the loading and advertising states, an OCF number tagged with the run directive causes routers to run the associated routing algorithms to generate forwarding tables. The resulting forwarding tables, along with the specified packet filters, are installed into the router.

The list of OCFs can change from one OCM to the next. In fact, this is precisely how the Operator introduces new OCFs into the system, and prepares the network to run them. A careful Operator will iterate a particular OCF through the loading and advertising stages, ensuring the expected operation before switching it to the run state. Old OCFs that are removed in succeeding OCMs are purged from the system, along with any algorithm files that they use.

3.2. OCF 0

There is one OCF that is considered immutable—OCF 0. All entities must advertise OCF 0 properties regardless of what other OCFs are loaded or which one is running. This OCF contains the minimum amount of information necessary to maintain routing functionality, and is always operational.

An OCF 0 forwarding table is always built using SPF [24] (Dijkstra's algorithm [12]) with hop count as its metric. FLINT traffic is always sent using this forwarding table. By ensuring FLINT traffic is forwarded using a straightforward, reliable routing table, FIRE application traffic, including LDTP, should always be correctly routed, allowing OCFs and their associated files to be downloaded regardless of the state of (in)operation of the currently running OCF.

OCF 0 contains exactly four properties:

FIRE Metric
: The hop count for this entity used to build the SPF routing table for FIRE management traffic. It defaults to one for each router node and zero for each link and network node.

IP Addresses
: The set of IP addresses associated with this entity. For links, this is the set of IP addresses associated with the source interface, both canonical and aliases. For routers, this includes any stub hosts that are reachable through this node. Networks do not participate in this property.

FIRE Up
: This boolean value states that FIRE is currently running for this node or link. If set to false, no traffic is routed through this entity.

OCFs Loaded
: A list of the OCF numbers (other than zero) for which this entity is flooding SAs. Only router nodes participate in this property.

In addition to being used by the OCF 0 routing algorithm, these properties are also available to routing algorithms running at any other OCF number.

4. ALGORITHMS

After an algorithm's support files are downloaded, the code is loaded into an execution environment on the router. Our FIRE implementation invokes algorithms in a Java Virtual Machine (JVM). If the algorithm is being asked to generate multiple forwarding tables based on different properties, a separate JVM (each with the same algorithm code) is created for each instance. Because we expect SPF to be a frequently used algorithm, in addition to being an integral part of OCF 0 routing for FIRE packets, SPF is implemented as a built-in function rather than a downloadable applet.

4.1. Programming Interface

The FIRE algorithm programming interface is intentionally very simple. Whenever new information is inserted into the property repository by the reception of an SA, a snapshot of the repository is made and the routing algorithm is called. The algorithm's job is to generate a forwarding table from the snapshot of the repository.

The programming interface does not explicitly support incremental updates. Our reasoning is that requiring algorithms to support incremental updates is both unreasonable (some algorithms may not have a straightforward way to do incremental updates) and, as an added complexity, subjects them to additional bugs. However, FIRE does permit the JVM to preserve data structures across calls to build a routing table, so programmers are free to perform incremental updates (or just cache useful information) if they wish. The programming interface is discussed in more detail in section 8.1.

4.2. Algorithm Frequency

When determining how frequently to run an algorithm, the key issue is correctness of routing. The fundamental idea behind link-state routing is that if everyone has the same information (the flooding protocol ensures information at all nodes will converge) and runs the same algorithms, they will get the same results and routing tables will be consistent. Obviously the sooner one runs algorithms in response to updates, the faster the convergence and the less likely that routing loops or black holes will occur. (It is instructive to consider the alternative. Suppose one ran algorithms only once every n minutes. If an SA arrives at one router just before it runs its algorithms, and at a neighboring router just after it runs its algorithms, the two neighbors will be out of sync for up to n minutes and traffic may loop.) In our view, loop freedom and black hole avoidance is vital to proper operation, so FIRE runs algorithms whenever new information arrives.

4.3. Thrashing

Given FIRE's predilection for invoking routing algorithms, it becomes important to protect against thrashing, where any new piece of information could cause an algorithm to run. FIRE tries to avoid thrashing in three ways:

First, algorithms cannot be stopped in mid-run. They run to completion with the property snapshot they have, and if an update is received while running, the algorithms are simply invoked again as soon as they complete. So FIRE algorithms are guaranteed to generate forwarding tables, regardless of the rate of incoming property updates.

Second, FIRE dallies slightly before invoking an algorithm. Rather than starting up each algorithm as soon as one new or updated SA arrives, FIRE waits a brief, configurable period (usually a few seconds) to allow additional new information to arrive, since routing updates tend to come in bursts.[2] Indeed, conventional wisdom holds that routing protocol traffic tends to be either very heavy (lots of new SAs) or very light (very few SAs)

[2] We have not found a careful study that discusses this behavior within an autonomous system. Chinoy's study [10] of backbone advertisements supports the idea that updates are bursty.

at any given moment. Dallying tries to ensure that during periods of heavy traffic, algorithm runs balance responsiveness with efficiency.

The final protection is not on algorithms themselves, but instead results from FIRE's limits on SA frequency. Since SAs can only be issued at a specified maximum rate (which is enforced by neighboring routers as part of the flooding protocol), a particular node cannot trigger system-wide algorithm runs too frequently. This mechanism is discussed in more detail in section 6.3.

5. PROPERTIES

Any information needed by routing algorithms to construct forwarding tables must be distributed throughout the network. FIRE packages this information into typed values called *properties*. Properties differ from metrics used in traditional routing algorithms. Metrics are weights, assigned to a link, that influence the likelihood of its inclusion in the path selection algorithm. Properties, on the other hand, are applicable not only to links, but to routers and networks as well.

For example, consider a network where the Operator wants to encourage a certain class of packets to be forwarded over the fastest links. In a metric-based system, the Operator would have to assign a weight to each link, where the weight corresponded in some fashion to the link's speed. In a property-based system, the link advertises its bandwidth as a property and the routing algorithm uses the actual bandwidths to find the fastest paths. Since bandwidths are bottleneck limited, the metric-based algorithm would likely be a modified version of SPF or a max-flow optimization.

Another example is the implementation of Core-Based multicast Trees [6]. Selecting optimal cores in a metric-based routing protocol is an open problem. In a property-based system, however, routers can dynamically advertise the property that they are willing to be cores. Properties can be thought of as name-value pairs, where the value is typed and can have multiple parts. So, for instance, routers might support a property named "multicast-core" whose value is a list of multicast groups for which the router is currently a core. A clever routing algorithm would then search the potential cores for one that was well-placed for the intended set of multicast recipients. This could be enhanced by having routers advertise how heavily they are loaded (e.g., the number of groups for which they are already serving as cores) and factoring load into the core selection algorithm.

The OCF defines the set of properties that a node or link must advertise. Some of these properties will make sense for both nodes and links, some will make sense only for nodes or links, and some will makes sense for only some nodes or some links. The OCF's grammar allows the Operator to specify the class of entity that should participate in advertising a particular property. If a participating entity is unable to generate a value for a particular property (perhaps it does not have the required hardware to support routing based on that property), the property's value can be set to *unsupported*.

Links are abstractions and cannot themselves issue SAs. The same is true for network nodes. The node that is the originating endpoint of a link is responsible for advertising the link. For network nodes, the Designated Router undertakes the responsibility of advertising for the network node and, in addition, for the links going from the network node to adjacent nodes as well. Note that this implies all network properties must either be statically configured or can be measured by an adjacent router.

A property can be generated by (a) using a configured value, (b) obtaining information from the router's MIBs, or (c) running a property applet. Configured values and MIBs are assumed to be in place prior to the circulation of an OCF containing algorithms that rely on these values.

5.1. Property Applets[3]

The ability to generate dynamic properties is one of the most powerful aspects of FIRE, as well as its most dangerous. FIRE borrows from the Active Networks [16, 36] philosophy, allowing downloaded code to be executed on the router itself. Unlike algorithms, however, which simply compute a function over the provided property repository, property applets need access to a far larger set of capabilities, possibly including file and network access. On the other hand, property applets, unlike more general dynamic router extensions, such as PLANet's *active extensions* [16], simply gather information about the relevant FIRE entity, and do not act in any way on the traffic being forwarded by the router. Even so, applets clearly represent a potential security risk. In addition to requiring all downloaded code to be cryptographically signed by a software authority (as discussed in section 7), our FIRE implementation uses Java's security infrastructure [15] to provide a balance between code security and applet functionality. The FIRE specification also allows for implementations to provide support for additional execution environments. The large body of work on Proof-Carrying Code [27] could be leveraged for installations with particularly tight security constraints.

Regardless of language or execution environment, property applets are constrained in what they can do. They can communicate with the local router on which they run, and they can send packets to the router's neighboring nodes (but no further, as routers need only advertise properties related to themselves, adjacent links, or directly-connected networks). Other than these basic restrictions, however, applets are allowed to execute arbitrary Java instructions. It is left up to the software signing authority to ensure that approved property applets function appropriately for use in a production environment.

5.2. Property Updates

Scheduling property applets is another difficult task. The operator may schedule applets to be run at specified intervals, or triggered by particular events, as specified by the OCF. Applets must be run at discrete times, however, and the processes the applets are trying to capture may not be discrete in nature. Furthermore, even if applets were run continuously, some control must be placed on the advertisement of new values. But FIRE cannot, in general, determine when properties have changed, since property values and types are arbitrary. Therefore property applets themselves are charged with notifying FIRE when their generated values have changed, and to what degree. The interface used for this notification is discussed in section 8.2.

FIRE uses the applet notification mechanism to determine when to issue new SAs. In the absence of explicit notification, FIRE issues new SAs periodically, at some configurable interval, usually on the order of minutes. If, however, one or more property values have recently changed, FIRE will schedule new SAs to be issued at a rate commensurate with the configured maximum SA rate. If an applet has indicated that a property value has changed dramatically enough to warrant immediate notification, an SA is issued immediately, subject to the flapping rules discussed in section 6.3.3.

---

[3] Note that neither routing algorithms nor property applets are *applets* in the strict Java sense of inheriting from the *java.applet.Applet* class.

6. PEERING & STATE DISTRIBUTION

Every routing protocol needs a mechanism to discover adjacent routers, termed *neighbors* or *peers*, and reliably distribute state information to them. Developing secure, robust mechanisms to support these functions can be quite difficult. Many previous routing protocols have been plagued by limited functionality, such as neighbor discovery algorithms that do not support uni-directional links [25], or buggy implementations [31].

Because of the difficulties in implementing state distribution and peering protocols, we decided not to make these functions programmable. Rather, FIRE fixes these essential mechanisms as built-in (non-programmable) infrastructure. Routing algorithms running on top of FIRE need not concern themselves with the subtle details involved in convergent, soft state distribution. FIRE builds on a considerable body of prior work [20, 25] to provide mechanisms that are secure, efficient, and robust.

6.1. FLINT

All routing traffic in FIRE is transported using FLINT. FLINT provides a multiplexing datagram service, similar to UDP, but under a separate protocol number. Using a special transport protocol for routing traffic provides several benefits. First, it provides a unique port space, free of the special implications of UDP and TCP port numbers. Second, it greatly simplifies the policies needed for IPsec's Security Policy Database. According to FIRE's security model, all FLINT traffic must be authenticated using IPsec (see section 7).

Each FIRE instance operates within a 128-port segment of the FLINT port space. The first port in this segment is known as the *base port*, and the ports used for the various FIRE protocols in each instance are derived by adding a specified offset from the base port. Using this scheme, multiple FIRE instances (possibly overlay networks, such as VPNs) can operate on the same physical links without the need for additional tunneling or encapsulation.

6.2. Neighbor Discovery

Topologically, FIRE models a network as a mesh of *nodes* connected by *links*. Besides all participating routers in the network, broadcast subnets are also treated as nodes, as in OSPF [25], thereby reducing the number of links between routers on a broadcast subnet from $O(n^2)$ to $O(n)$.

Unlike OSPF, however, FIRE explicitly supports uni-directional links. Links are defined as uni-directional, so a bi-directionally connected pair of nodes has two links between them, one in each direction. Two nodes are considered neighbors if some combination of adjacencies (that does not pass through another router node) supports bi-directional communication between the two nodes—a two-way link is not required.

The sample network in Figure 2 contains six nodes: three routers and three broadcast networks. The solid arrows represent directed adjacencies. Suppose Networks 1 and 3 were Ethernets, and Network 2 was a satellite network. Since FIRE models links as uni-directional, there are two links between Router A and Network 1. The same is true between Router B and Network 3, and Network 3 and Router C. In this example, however, Router A has the only uplink to Network 2, while all routers have downlinks. The dashed lines depict the peering relationships that would be established in this topology. Routers B and C peer in the standard fashion across Network 3. Routers A and B also form a peering relationship, even though they must use two separate physical links in order to communicate. In contrast, Routers A and C do not peer, as no single-hop path exists from C to A.

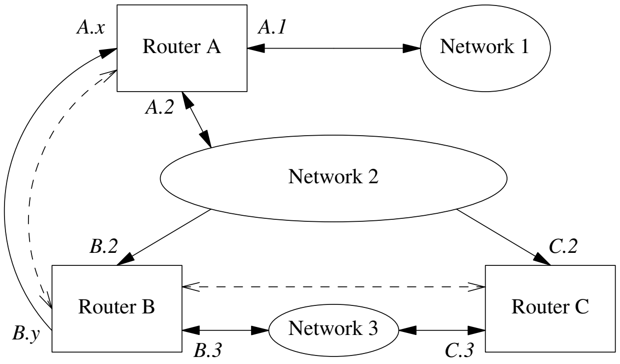

Figure 2: FIRE Network Model

The FIRE Peering Protocol is based largely upon the OSPF's Hello Protocol [25]. It establishes peering relationships between adjacent routers for the exchange of SAs. It also selects a *Designated Router* (DR) and *Backup Designated Router* (BDR) for each broadcast subnet.

6.2.1. Designated Routers

Designated Routers serve two purposes. First, they are responsible for issuing property updates for the subnet and its associated links. Second, they help to limit the number of peering relationships established over a particular subnet. Routers on a broadcast subnet peer only with the DR for the network, rather than having to peer with all routers on the subnet.

Routers functioning as DRs in FIRE must have a bi-directional connection (possibly using two separate interfaces) to the network they are representing. This restriction is enforced by the FIRE Peering Protocol, as discussed below. In the case of wireless broadcast networks, there may be no guarantee that a single node can communicate with all other FIRE nodes on the wireless subnet. If there exists no node that can reliably broadcast to all the others, the wireless network must be modeled (for the purposes of FIRE peering) as a mesh of point-to-point links, rather than as a broadcast medium. The same is true for non-broadcast multi-access networks (NBMA) networks.

To prevent flapping of the DR (and hence repeated re-issuance of the network properties for which the DR is responsible), the Peering Protocol also elects a BDR. If the DR ever fails, the BDR assumes the role of the DR, and a new BDR is elected. If two DRs (or BDRs) are ever present on a single subnet due to network healing, one of the candidates is selected based upon election rules very similar to those of OSPF, with one caveat. Since DRs may be required to run property applets on behalf of the subnet, FIRE's election process ensures the DR has a bi-directional connection to the subnet.

6.2.2. FIRE Peering Protocol

Every router is configured with a set of interfaces over which FIRE peering relationships (which may not be the same as the list of interfaces over which traffic is routed) are formed. FIRE is then changed to manage these interfaces in a manner very similar to OSPF.

The Peering Protocol is essentially a beaconing process, in which all participating routers send out beacon packets at jittered regular intervals. These beacons contain information about the originating router's FIRE ID, and the sending interface's IP address, netmask, and priority. Beacons sent over broadcast media may also contain flags indicating the originator believes itself to be either the DR or BDR for that subnet. Beacons are constructed per interface, and multicast to a well-known multicast IP address on a well-known offset from the base port.

6.3. State Distribution

Once a peering relationship has been established between two or more neighboring routers, they begin exchanging routing information through the use of SAs. These SAs are reliably flooded throughout the network, thereby providing a robust, convergent method of distributing shared state across the network.

6.3.1. State Advertisements

SAs contain information about the AS in which FIRE is running. There are four types of SAs: configuration, certificate, external route, and property. *Configuration SAs*, also known as Operator Configuration Messages (OCMs), are generated by the operator and are detailed in section 3. *Certificate SAs*, as discussed in section 7, are used to distribute public keys and authority certifications. Routes managed by protocols other than FIRE (such as exterior routes) are advertised through the use of *External Route SAs*.

Property SAs advertise an entity's metrics for a particular OCF. Each node and link has an associated property SA for each OCF being advertised. The particular properties listed are determined by the OCF and the type of the entity being advertised. The payload of a property SA is a self-parsing S-expression containing values for each property. Special flag values are available to define a property as being unsupported, or that a particular entity is a non-participant.

SAs contain a *creator* field which contains the FIRE ID of the originating entity in addition to an *identifier* field which identifies the entity being described in the SA. For property SAs associated with a router, the creator and identifier fields are identical. Property SAs for links are originated by the router on the transmission end of the link; DRs issue property SAs for their respective subnets and any links the subnets may be responsible for.

An SA is uniquely identified by its type, identifier, and OCF number fields. Every SA is also timestamped and has a sequence number, so SAs with the same type, identifier, and OCF values can be ordered.

6.3.2. SA Refreshment

In addition to being timestamped, each SA is also given an expiration time, after which it is considered invalid, and is no longer flooded by FIRE. To prevent instability caused by expiring SAs, routers periodically renew SAs they have generated before the previous version expires. To reissue an SA, the router first replaces the superseded SA in its own SA cache with the newly generated version, and then floods the new SA to each of its neighbors.

6.3.3. Damping of Flapping

Whenever an attached interface comes up, the router must advertise its existence by issuing SAs. In order to prevent *flapping*, the rapid re-issuance of SAs for the same entity, we utilize the Skeptic model from Autonet [30]. When a new SA is warranted, due to a change in the properties contained within, the Skeptic delays slightly before issuing the SA. It not only limits the rate of SA issuance to a fixed maximum rate, but penalizes rapidly changing SAs by exponentially increasing the delay with each new request. As request frequency decreases, the Skeptic reduces the delay penalty accordingly. However, if a router determines an associated link has gone down and was previously advertised as being up, it immediately generates a new SA for that link indicating the link has gone down. This rule insures inoperable links are always eliminated from the topology.

6.4. Reliable Flooding

Each FIRE router maintains a cache of all current SAs. The purpose of the state distribution mechanism is to maintain a consistent shared view of the set of current SAs across all routing nodes. Reliable flooding is employed to make sure that an SA generated by one node is eventually received by all nodes in the AS. Stated simply, each router is responsible for forwarding any new SA to all of its neighbors. To prevent SAs from being unnecessarily flooded to neighbors that have already indicated they have a copy, either by preemptively acknowledging it, or by actually sending the SA itself, routers maintain state for each neighbor associated with every SA in its cache.

6.4.1. State Message Transmission

Information is exchanged between FIRE nodes using State Messages, sent to a well-known offset from the base port. A State Message contains either an SA or an acknowledgment of the receipt of one or more SAs. State Messages are flow controlled using a simple windowing protocol. The transmission window is specified in terms of the number of SAs that may be transmitted without acknowledgment. Each FIRE implementation has a fixed, built-in window size (the default is eight), which is the same for all FIRE routers in an AS. A router maintains separate transmission queues for each neighbor. SA acknowledgment messages are never queued, but instead are sent as soon as possible. The remaining SAs are queued and transmitted as soon as window space is available.

When possible, messages being flooded to multiple neighbors reachable via the same interface are multicast. A threshold value (four is the default) specifies a portion of the receive window that is reserved for multicast traffic on multicast-enabled receive interfaces. After the threshold is reached, only multicast packets may be sent to the receiver until some of the unicast packets have been acknowledged. This threshold prevents a large amount of traffic intended for one neighbor from delaying the delivery of multicast traffic to the group.

6.4.2. Retransmission

To provide a reliable flooding mechanism, FIRE retransmits unacknowledged State Messages in the transmission window at intervals based upon the round trip time (RTT) between neighbors, as measured using Jacobson's algorithms [18]. Successive retry attempts are backed off exponentially until either an acknowledgment is received or a maximum retry count is exceeded. However, unlike initial floods, retransmissions are unicast directly to specific neighbors rather than multicast.

6.4.3. State Message Processing

Upon receipt of a State Message, a router first looks into its cache to see if it has already been received. If so, it simply acknowledges the message and completes processing. If, on the other hand, the message contains an SA the router has not seen before, it firsts validates that the SA header is properly formed. If the header is invalid for whatever reason, the router immediately sends an acknowledgment to the sender, echoing back the header information.

Routers receiving acknowledgments compare the enclosed header with outstanding transmissions. If the SA was damaged in transmission, the headers will not match, and the SA will be retransmitted as if unacknowledged. If, however, the corruption occurred in the sender itself (hence any retransmissions would be equally useless), the acknowledgment will cause the offending router to cease its transmissions of the corrupted SA.

Assuming a received SA has a well-formed header, a router then verifies the signature before sending an acknowledgement, caching the SA, and flooding the SA to its neighbors. In the case of a non-DR router on a broadcast subnet, it has only one neighbor on that subnet: the DR. Only the DR floods SAs to every member of the subnet.

6.5. Synchronization

Reliable Flooding distributes SAs to all routers currently connected to the network. When new routers come up, however, or disjoint portions of the network are reconnected, the SA repositories must be resynchronized. This synchronization is done through the *State Dump* process.

The State Dump process is initiated whenever a new adjacency is formed. Since the adjacency may form asynchronously due to the beaconing process of the Peering Protocol, the start of the dump is delayed for some period or until the adjacent router initiates it. The State Dump procedure seeks to swiftly synchronize two neighbors while generating the near minimum amount of traffic. No special message types are used by the process; state dumps use SAs and acknowledgements so that if new SAs are sent during the State Dump process, the SAs can simply be integrated into the State Dump data stream.

A peer-to-peer process, both routers perform an identical set of functions. Each sends an acknowledgment for every SA in its cache to the new neighbor. Since the number of SAs to be acknowledged may be large, it may be necessary to spread them across multiple State Messages. This bulk acknowledgment serves as a synopsis of the contents of each router's SA cache and allows the adjacent router to note which SAs need not be transmitted. The router then places the Configuration SA (if available) into the send queue. It follows these SAs with the remaining unacknowledged SAs in its cache.

As each router receives SAs and acknowledgments from its neighbor, the flooding procedure causes the removal of corresponding SAs from the send queues. Because acknowledgments are sent first and SAs that are acknowledged are not transmitted, most of the SAs that are sent between the neighbors will be new.

6.6. A Bootstrapping Example

Figure 3 shows an annotated `tcpdump` trace of the packet exchange generated by three FIRE routers being simultaneously brought up on the same subnet.[4] The trace is shown in relative time, with the whole process completing in just over a minute and a half. The delay is dominated by the configurable wait imposed by the Peering Protocol for DR election and dally preceding synchronization. As can be seen in the *Neighbor Discovery* portion at the beginning of the trace, each router issues a peering beacon on boot-up, announcing its presence on the subnet. At the next beaconing interval (10 seconds in this example) each router advertises that it has heard from the others, also including the interfaces the received beacons were sent and received on.

This beaconing process continues on indefinitely. We have elided further peering beacons from the trace for the sake of

[4] For clarity, neither IPsec (including LGKP) nor certificate exchange are enabled in this example.

Neighbor Discovery

```
00.3233 grumpy.1051 > OSPF-ALL.2080:
    grumpy grumpy mask 255.255.0.0 sn 0 pri 10
00.9918 sleepy.1036 > OSPF-ALL.2080:
    sleepy sleepy mask 255.255.0.0 sn 0 pri 10
01.7104 happy.1031  > OSPF-ALL.2080:
    happy  happy  mask 255.255.0.0 sn 0 pri 10

10.3324 grumpy.1051 > OSPF-ALL.2080:
    grumpy grumpy mask 255.255.0.0 sn 1 pri 10
    happy  happy  grumpy, sleepy sleepy grumpy
10.9990 sleepy.1036 > OSPF-ALL.2080:
    sleepy sleepy mask 255.255.0.0 sn 1 pri 10
    grumpy grumpy sleepy, happy  happy  sleepy
11.7188 happy.1031  > OSPF-ALL.2080:
    happy  happy  mask 255.255.0.0 sn 1 pri 10
    sleepy sleepy happy,  grumpy grumpy happy
```

B/DR Election

```
60.3347 grumpy.1051 > OSPF-ALL.2080:
    grumpy grumpy mask 255.255.0.0 sn 6 pri 10 DR
    happy  happy  grumpy, sleepy sleepy grumpy
60.9970 sleepy.1036 > OSPF-ALL.2080:
    sleepy sleepy mask 255.255.0.0 sn 6 pri 10
    grumpy grumpy sleepy, happy  happy  sleepy
61.7278 happy.1031  > OSPF-ALL.2080:
    happy  happy  mask 255.255.0.0 sn 6 pri 10 BDR
    sleepy sleepy happy,  grumpy grumpy happy
```

Network SA Generation

```
62.3421 grumpy.2096 > OSPF-ALL.2096: ocf0 192.1/16-NET
62.3425 happy.2096  > grumpy.2096: ocf0 192.1/16-NET ack
62.3426 sleepy.2096 > grumpy.2096: ocf0 192.1/16-NET ack

62.3439 grumpy.2096 > OSPF-ALL.2096: ocf0 192.1/16-happy-LNK
62.3441 happy.2096  > grumpy.2096: ocf0 192.1/16-happy-LNK ack
62.3442 sleepy.2096 > grumpy.2096: ocf0 192.1/16-happy-LNK ack

62.3455 grumpy.2096 > OSPF-ALL.2096: ocf0 192.1/16-sleepy-LNK
62.3458 happy.2096  > grumpy.2096: ocf0 192.1/16-sleepy-LNK ack
62.3459 sleepy.2096 > grumpy.2096: ocf0 192.1/16-sleepy-LNK ack

62.3472 grumpy.2096 > OSPF-ALL.2096: ocf0 192.1/16-grumpy-LNK
62.3474 happy.2096  > grumpy.2096: ocf0 192.1/16-grumpy-LNK ack
62.3475 sleepy.2096 > grumpy.2096: ocf0 192.1/16-grumpy-LNK ack
```

State Dump

i) synchronization

```
90.3513 grumpy.2096 > sleepy.2096: ocf0 grumpy-RTR ack [|fire]
90.3518 grumpy.2096 > happy.2096:  ocf0 grumpy-RTR ack [|fire]
90.3520 sleepy.2096 > grumpy.2096: ocf0 sleepy-RTR ack [|fire]
90.3524 happy.2096  > grumpy.2096: ocf0 happy-RTR  ack [|fire]
```

ii) grumpy — sleepy

```
92.3621 grumpy.2096 > sleepy.2096: ocf0 grumpy-RTR
92.3628 sleepy.2096 > grumpy.2096: ocf0 grumpy-RTR ack
92.3623 grumpy.2096 > sleepy.2096: ocf0 grumpy-192.1/16-LNK
92.3639 sleepy.2096 > grumpy.2096: ocf0 grumpy-192.1/16-LNK ack
92.3625 sleepy.2096 > grumpy.2096: ocf0 sleepy-RTR
92.3657 grumpy.2096 > sleepy.2096: ocf0 sleepy-RTR ack
92.3623 sleepy.2096 > grumpy.2096: ocf0 sleepy-192.1/16-LNK
92.3635 grumpy.2096 > sleepy.2096: ocf0 sleepy-192.1/16-LNK ack
```

iii) grumpy — happy

(essentially identical to step ii, above)

DR Flooding

```
92.3641 grumpy.2096 > OSPF-ALL.2096: ocf0 sleepy-192.1/16-LNK
92.3647 happy.2096  > grumpy.2096: ocf0 sleepy-192.1/16-LNK ack
92.3650 sleepy.2096 > grumpy.2096: ocf0 sleepy-192.1/16-LNK ack

92.3663 grumpy.2096 > OSPF-ALL.2096: ocf0 sleepy-RTR
92.3665 sleepy.2096 > grumpy.2096: ocf0 sleepy-RTR ack
92.3669 happy.2096  > grumpy.2096: ocf0 sleepy-RTR ack

92.3683 grumpy.2096 > OSPF-ALL.2096: ocf0 happy-RTR
92.3685 happy.2096  > grumpy.2096: ocf0 happy-RTR ack
92.3686 sleepy.2096 > grumpy.2096: ocf0 happy-RTR ack

92.3702 grumpy.2096 > OSPF-ALL.2096: ocf0 happy-192.1/16-LNK
92.3704 happy.2096  > grumpy.2096: ocf0 happy-192.1/16-LNK ack
92.3705 sleepy.2096 > grumpy.2096: ocf0 happy-192.1/16-LNK ack
```

Figure 3: FIRE Bootstrap Trace

brevity. Roughly a minute after boot-up (recall that all periodic events are jittered slightly), each router independently conducts a DR election, and the winners announce the results in their beacons, as can be seen in the traces labeled *B/DR Election*. The newly elected DR, `grumpy`, then proceeds to advertise SAs for the network, multicasting them to all routers on the subnet.

Finally, approximately 30 seconds after DR election, the *State Dump* procedure is initiated between each router and the DR. The independent processes are roughly identical, hence the packets for the second dump are not included in the trace. After synchronizing with each router, the DR then floods the new SAs from each router to the remaining router(s). The DR multicasts these SAs for efficiency, hence the issuing router also receives a duplicate copy, which it ACKs and silently discards. SAs for all ten entities on the subnet (3 routers, with two links a piece, and the network itself) are distributed in only 16 messages. In general, a subnet with n routers ($3n + 1$ entities) will exchange $7n - 5$ messages.

7. SECURITY

Implementing a security infrastructure for a routing protocol presents an interesting problem. To provide most security services, one needs a key infrastructure. But generic key infrastructures generally require pre-existing packet routing functionality. Our solution is to implement any security infrastructure that FIRE requires within FIRE itself.

7.1. Attacks

A routing protocol, FIRE in particular, is subject to attacks of two basic types:

Wiretapping

Attackers are assumed to have access to the communications links in such a way that they may modify, suppress, insert, or replay FIRE messages or fragments. FIRE must therefore protect against such attacks. Insertion of bogus fragments could prevent a re-assembled message from being accepted at the destination; replaying old messages might disrupt current activities (such as electing a DR); flooding a router with bogus FIRE messages or tampering with data in FIRE messages could result in a denial of service.

Subversion

The routers and other FIRE nodes may be subverted, either physically such that an attacker completely controls a router's behavior or by use of compromised keying material such that an attacker may originate messages that appear to come from the router. A subverted node could send out inaccurate data, possibly affecting a much larger portion of the network.

Many of the harmful behaviors described could also occur due to bugs in software or hardware. Thus, FIRE's design expects that a certain amount of misbehavior (intentional or not) will occur. FIRE's security model is built on a philosophy of containment: our goal is to bound the effects of misbehavior and detect the misbehavior whenever possible.

FIRE meets this goal using three sets of mechanisms. First, FIRE employs a certificate infrastructure with a tiered authority structure. These certificates advertise the public keys of nodes, links and entities such as the Operator. All SAs are digitally signed to provide end-to-end authentication and integrity. Second, FIRE makes use of IPsec to protect against certain hop-by-hop attacks that end-to-end security measures cannot prevent. IPsec's authentication, data integrity, and anti-replay services make it very difficult for a non-participating entity to inject FIRE traffic. Third, and most importantly, the FIRE protocols are designed to be robust against the failure or subversion of an individual router or set of routers.

7.2. Certificates and Digital Signatures

FIRE's basic security mechanisms are based on public key cryptography, using X.509 [1] certificates, and patterned after the existing work on Secure OSPF [26] and Secure BGP [20]. Each FIRE node participates in a certificate hierarchy that is managed by the FIRE system. At the top of the hierarchy is the Root. Each separate FIRE instance has its own certificate hierarchy, therefore different ASs will have unique Roots. Routing information shared at border routers must be secured by the exterior gateway protocol.

The Root creates certificates for a set of principals that are entitled to perform various actions. These principals include the Operator and a Software Master. The Operator is the logical entity that runs the network. The Software Master is the logical entity that approves algorithm and property applets (the choice of which approved programs are used is left to the Operator). Below the Operator sit the FIRE nodes themselves. Each node has its own public/private key pair and certificate. FIRE certificates are circulated as part of the normal FIRE flooding protocols in special Certificate SAs. Each node is responsible for flooding its own certificates.

Each SA is signed by its creator. Thus, even though almost all FIRE messages are relayed through other nodes, messages are protected from tampering in-flight. Because each message is unambiguously linked to its signer, advertising of false information is limited and can be traced. A node can only lie about the information it is entitled to advertise.

While digital signatures solve many security problems, there is still a class of attacks they do not protect against. These hop-by-hop attacks are dealt with by IPsec and the FIRE protocols themselves, as discussed next.

7.3. IPsec

FIRE uses IPsec to protect single hop links between nodes. In the absence of protection by IPsec, a non-FIRE entity could replay FIRE messages. Furthermore, because peering must be negotiated before FIRE certificates are exchanged, the Peering Protocol needs protection against replay and corruption of packets. IPsec's ESP header [21] with anti-replay protection guards against these attacks.

One challenge in supporting IPsec is that FIRE uses multicast to optimize local communications. IPsec lacks a protocol to efficiently distribute keying material for local multicast groups, so we developed a lightweight protocol, the Local Group Keying Protocol (LGKP), that distributes symmetric keying material to all participating routers on a subnet.

7.4. Protocol Robustness

Some security features are implemented by the FIRE protocols themselves. Most notably, FIRE implements reliable flooding. Reliable flooding ensures that if one uncompromised path exists between a creator of a message and a consumer, the consumer will eventually see the message. Reliable flooding achieves this guarantee by (a) allocating buffering in each node on a peer-by-peer basis, so no one peer can consume all the buffering in a node and cause a denial-of-service attack that prevents a message from being relayed; and (b) flooding every message over all links.

Unlike many previous routing protocols, FIRE's flooding mechanism does not utilize any explicit request messages. The avoidance of any such *pull* mechanisms prevents denial-of-service attacks launched by malevolent routers that continue to request already-received information from adjacent routers.

In addition to ensuring the reliable dissemination of routing information, FIRE also limits the spread of disinformation. The information that a node may advertise is limited by a signed set of permissions contained in an X.509 attribute certificate. Thus, particular routers can be restricted to advertising links only to adjacent networks, preventing a subverted router from being able to black-hole arbitrary traffic.

Other FIRE protocols also have protective mechanisms. LDTP has mechanisms to protect against attacks similar to SYN-flooding, where an attacker creates multiple partial sessions that tie up resources at the LDTP server. Similarly, the Peering Protocol contains features that make it difficult for routers to cause the DR to change unless the existing DR goes down.

8. PROGRAMMING INTERFACES

One of the key features of FIRE is the ability to dynamically load new routing algorithms and to define new properties using applets. Ideally, FIRE implementations would use a language designed to support self-proving applets (programs that could be verified to meet certain constraints). By making it possible to verify (within limits) how a program behaved, the risk of software bugs could be dramatically reduced. Our reference implementation, however, uses Java, both because the self-proving languages were not mature enough (not one had a stable virtual machine that could be embedded into the FIRE implementation) and because Java's popularity reduces the learning curve for FIRE algorithm and applet developers.

The choice of Java, however, forced us to pay more attention to the design of FIRE's programming interfaces. Writing graph algorithms is a notoriously difficult problem—bugs are common. This difficulty is compounded by Java's support for a range of complex language features such as concurrency and event handling, which also tend to produce programmer bugs. FIRE's programming interfaces a deliberately simple, in an effort to encourage a straightforward programming style.

```
public interface Algorithm
{
    // initialize the class
    public void init(String[] args);

    // generate forwarding table entries; will be called multiple
    // times; state will be stored across invocations
    public void run(Repository repo);

    // cleanup when we're done
    public void cleanup();
}
```

Figure 4: Java Algorithm Interface

8.1. Routing Algorithm Interfaces

A routing algorithm is provided with two separate programming interfaces: the algorithm interface and the forwarding table API. Every algorithm must implement the Java public interface shown in figure 4. The algorithm is invoked by FIRE through calls to this interface.

After an algorithm's code files are loaded, a separate instance of the JVM is created for each invocation of the algorithm and the *init()* function is called with an array of strings specified in the OCF. (The separation of each algorithm into a different instance of the JVM allows any particular routing algorithm to fail without halting the proper operation of the remaining algorithms supported by the router.) The argument strings (similar to *C*'s `argv`) may be used to pass arbitrary operator-specified values into the algorithm, such as threshold values. The role of *init()* is to do any initialization required and prepare the environment for forwarding table generation.

	OCF 0				OCF 1	
FID	FIRE Metric	IP Addresses	FIRE Up	OCFs Loaded	Delay (ms)	Drop %
A	1	A.1, A.2, A.x	*true*	1	*NP*	2%
A → 1	0	A.1	*true*	*NP*	2	0%
1	0	*NP*	*true*	*NP*	*NP*	*NP*
1 → A	0		*true*	*NP*	2	0%
A → 2	0	A.2	*true*	*NP*	250	*unsup*
2	0	*NP*	*true*	*NP*	*NP*	*NP*
2 → A	0		*true*	*NP*	250	3%
2 → B	0		*true*	*NP*	250	3%
2 → C	0		*true*	*NP*	250	3%
B	1	B.2, B.3, B.y	*true*	1	*NP*	0.5%
B → A	0	B.y	*true*	*NP*	40	1%
B → 3	0	B.3	*true*	*NP*	1	0%
3	0	*NP*	*true*	*NP*	*NP*	*NP*
3 → B	0		*true*	*NP*	1	0%
3 → C	0		*true*	*NP*	5	0%
C	1	C.2, C.3	*true*	1	*NP*	0%
C → 3	0	C.3	*true*	*NP*	5	0%

Figure 5: A Property Repository

FIRE periodically invokes the algorithm's *run()* method on a snapshot of the repository for the relevant OCF. Figure 5 shows a sample property repository for the network in figure 2. The repository contains two properties the operator has selected for OCF 1, in addition to the OCF 0 properties. Adjacencies can be deduced from FIRE IDs (FIDs), as can the type of entity. Entities not participating in a particular property are denoted as *NP*.

An algorithm's implementation has built into it property names to use as input. The OCF specifies a mapping from currently advertised property names to the names required by the routing algorithm. OCF 0, for example, maps *FIRE Metric* to the name expected by the built-in SPF algorithm in order to compute routing tables for FLINT traffic. If the operator desires to run the algorithm on a different property in another OCF, a new name mapping would be used. If, on the other hand, more sophisticated preprocessing is desired, such as EIGRP's cost function, a wrapper function can be employed to read the existing repository and rewrite the desired metrics into the appropriate property, before passing the repository on to the standard routing algorithm.

At the end of a run, a routing algorithm must generate a forwarding table, using the API shown in figure 6. Again the API is deliberately simple. The forwarding table is created before the JVM is invoked, so the table is always present (no initialization is required). Furthermore, the interface does not distinguish between modifying and adding an entry. The algorithm simply states that it needs the following entry in the table, and the API will either modify the existing entry or add a new one as required. All updates are batched for efficiency, and changes to the forwarding table are only made when the *do_updates()* method is called.

```
public class ForwardingTable
{
    // update an entry; if the entry doesn't already exist, add
    // it; if it does exist, modify it
    public static native void update_entry(InetAddress destination,
                int mask_length, InetAddress next_hop, FID iface);

    // delete an entry from the forwarding table
    public static native void delete_entry(InetAddress destination,
                                        int mask_length);

    // purge all entries from the forwarding table
    public static native void purge_table();

    // apply all of the updates from the above calls
    public static native void do_updates();
}
```

Figure 6: Java Forwarding Table API

8.2. Property Applet Interface

The interface for property applets is similar to that for algorithms. FIRE calls *init()* when the property is first specified by the current OCF and passing in the argument string from the OCF. As with routing algorithms, when the property applet is no longer in use, FIRE terminates it by calling *cleanup()* before shutting down the enclosing JVM.

Unlike algorithms, which respond to changes, the goal of property applets is to detect changes. One might imagine this difference would result in very different ways of invoking them. While it would be desirable to allow applets to specify a complex, event-driven mechanism to trigger themselves, such an implementation proved far too complex and mistake-prone. Instead, the applet invocation interface is essentially identical to that of algorithms; an applet is simply invoked periodically by calling *generate()*. The timing and frequency of FIRE's calls to *generate()* are specified in the OCF.

```
public class SA_update
{
    // report data back to FIRE
    public static native void report_data(Object value);

    // tell FIRE it should send an SA in the next cycle
    public static native void value_changed();

    // tell FIRE to send an SA ASAP
    public static native void force_SA();
}
```

Figure 7: Java SA API

When an applet is run, it may choose to record an updated value for the property or leave the existing value alone. For especially noisy properties, it may be desirable to squelch the vales within applet itself, rather than continually issuing SAs with different values. Furthermore, some changes may be significant while others are not (e.g., a change in measured queueing delay from 50 ms to 49 ms probably isn't very important, but a change from 50 ms to 10 ms likely is). FIRE itself has no idea what changes in values are significant, so the API shown in figure 7 allows the property applet to provide a notion of significance. The applet can simply record a new value, in which case the new value will be advertised only when the next SA is periodically generated (assuming the value is not updated again before the SA is generated). If the change is significant, however, the applet can call *value_changed()*, indicating that it would be desirable to send an SA with the new property value. Or the applet can indicate with *force_SA()* that the property's value has changed so dramatically that an SA should be sent immediately (subject to SA damping rules).

9. IMPLEMENTATION

We have implemented FIRE as a user-level daemon with supporting kernel modifications for FreeBSD. Multiple FIRE daemons may coexist to support separate FIRE instances on the same router (by communicating on different base ports). Each FIRE daemon, however, must be responsible for a disjoint set of traffic. This section discusses some of the more interesting features of the implementation.

9.1. Forwarding

FIRE places two new requirements on the IP forwarding mechanism. First, it requires every packet to be filtered. Second, it requires support for multiple forwarding tables. Both changes were novel for a BSD kernel.[5]

9.1.1. Packet Filtering

FIRE packet filters are specified using the Berkeley Packet Filter (BPF) [23] syntax, which provides a well-known and highly portable specification language. For performance reasons, we restrict FIRE BPF filters to the IP header. Even with this limitation, however, the BPF kernel implementation is not adequate for filtering at line speed. We chose to implement Dynamic Packet Filtering (DPF) [13] in the FreeBSD kernel with a device driver interface similar to BPF. Our kernel DPF implementation provides multiple filter sets, each of which is individually accessible using standard UNIX file system semantics (and permissions).

One filter set in particular, `/dev/fwf0`, is known as the *überfilter*. All IP packets forwarded by the router (by *ip_forward()*) are passed through the überfilter. If no match is found, the packet is forwarded according to the default kernel forwarding table (the default kernel table exits to support legacy applications that need access to the old kernel routing table). If, however, the packet matches a filter installed in the überfilter, it is passed through the specified secondary filter set. A matching filter in the secondary set specifies the appropriate forwarding table to use. If no match is found, the packet is dropped.

Each FIRE daemon requests its own filter set, and then installs a filter in the überfilter corresponding to the set of traffic it is responsible for, causing DPF to route any matching packets through the daemon's secondary filter set. Our implementation uses a translation library to install the BPF-style filters specified in the OCF directly into its secondary filter set, where DPF efficiently demultiplexes them to the appropriate forwarding table.

9.1.2. Forwarding Tables

Once the correct forwarding table has been selected, FIRE follows the standard IP forwarding mechanism and finds a best matching prefix for the destination address. Due to the additional performance cost of filtering, we replaced the standard BSD radix-trie lookup tables [32] ($O(W)$ performance, where W is the length of an address) with the ETH-WASHU lookup algorithm [35]. This algorithm both reduces the worst-case lookup to $O(log_2 W)$ and is more space efficient than the BSD algorithm.

---

[5] We note that in-kernel packet filtering itself is not new, and is used for efficient protocol demultiplexing in the Nemesis [22] and Scout [5] operating systems, among others.

Kernel forwarding tables are updated using the standard BSD *PF_ROUTE* socket interface; the only change is a forwarding table handle is now passed in `rt_msg`. The new forwarding table lookup function returns a `struct rt_entry` for the next hop router, which is backward compatible with routes returned by *rtalloc()*.

9.2. Sandboxing

The implementation currently supports JVML routing algorithms and property applets through the use of TransVirtual Technologies' Kaffe. Each algorithm or applet is run in its own Kaffe Java Virtual Machine (JVM), interfacing with FIRE through the use of the provided Java Native Interface (JNI) API discussed in section 8. The JNI functions are supported by a thin C wrapper that interfaces with both the FIRE daemon and the kernel. Forwarding table updates are communicated directly to the kernel by the wrapper through the use of a *PF_ROUTE* socket. In the case of algorithms associated with OCFs that are only *advertised*, and not yet *running*, the forwarding table functions are stubbed, and requested updates are logged, but not yet installed into the kernel forwarding tables.

Communicating property values is a bit trickier, however. Since the Property Repository is constantly in a state of flux, a static snapshot is provided to routing algorithms when they are invoked. This snapshot is passed to the JVM through the file system. The C wrapper reads the repository snapshot files and marshals them into appropriate Java structures as required by the algorithm. The FIRE daemon passes control messages to the JVM through a named pipe. Results of property applets are communicated back to the FIRE daemon in a similar fashion. Figure 8 depicts the interaction of the various parts of our FIRE implementation.

9.3. Performance

We have performed a preliminary performance analysis on our prototype implementation. Because FIRE is implemented in a FreeBSD kernel rather than a commercial router platform, the forwarding performance results are not particularly interesting. Our implementation forwards packets about 33% slower than stock FreeBSD, mostly due to deficiencies in our prototype code. We believe this performance gap would be almost entirely eliminated in an optimized implementation.

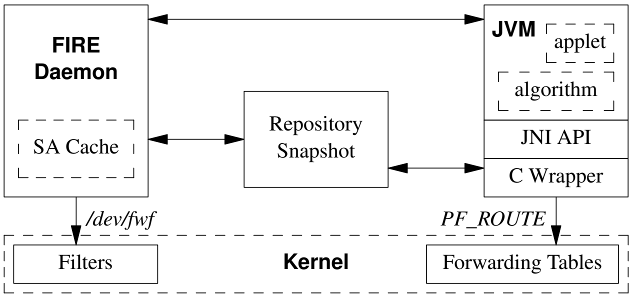

Figure 8: FIRE Daemon Implementation

Of more import is the performance of the routing protocol itself, measured in packet exchanges. As shown in figure 3, our FIRE implementation generates at most 2 packets (ignoring retransmissions due to packet loss) on a subnet for each entity advertised (one packet from the creating entity to the DR, and one multicast packet from the DR to the other nodes). ACKs are bundled together, hence the exact ACK count varies dramatically based on the precise timing of events. At OCF 0, which provides SPF functionality, our message count is a small multiple of OSPF (approximately $3x$, since, unlike OSPF, we count links and networks as separate entities). Each additional advertised OCF generates a similar number of packets, resulting in a linear growth in traffic. Note, however, that one OCF may contain properties for generating multiple forwarding tables, hence the incremental cost of additional forwarding tables is only in the size of the messages, not the packet count.

10. RELATED WORK

There has been considerable work over the past decade on supporting different types of service. The focus of the vast majority of this work has been on queueing schemes such as Fair Queueing [11], and schemes such as Guaranteed Service and Differentiated Services [28] that use these queueing schemes to support different service levels. This work, however, has not examined routing system support. FIRE is, therefore, complementary to these schemes, in that FIRE provides support for routing according to different requirements, but does not specify queueing regimes.

Other routing protocols have sought to be extensible. OSPF [25] allows the definition of new state advertisement messages (this feature was used for Secure OSPF). IS-IS [8] has similarly flexible features. The major distinction between these protocols and FIRE is that both protocols require recoding of existing implementations to generate and use the new information being advertised, while FIRE provides run-time support. This also contrasts with Multi-Protocol Label Switching (MPLS) [3], where an operator can manually configure paths through the network for specific classes of traffic. The MPLS solution is static, while FIRE, like most routing protocols, offers a completely dynamic solution.

While FIRE provides the ability to download and execute arbitrary code on production routers, FIRE is not a traditional *Active Network*—unlike [16, 36], user generated traffic cannot affect routing behavior in FIRE. This separation of operator-initiated control flow and forwarded data traffic allows FIRE to sidestep many of the difficult performance and security issues associated with Active Networking, although not all. In particular, FIRE's authentication and authorization scheme addresses similar problems as those dealt with in SANE [2]. In addition, basic router software verification techniques, such as SANE's secure bootstrapping, remain applicable.

11. CONCLUSION & FUTURE WORK

FIRE significantly increases the control network operators have over how their network routes traffic. Operators can change routing algorithms and metrics at run time and dynamically configure which traffic classes are forwarded by the various algorithms. By enabling network operators to change the routing algorithms employed by the protocol, FIRE significantly lowers the barrier to deploying new algorithms in an AS. FIRE also presents several interesting new research problems.

One obvious research problem is whether FIRE's concepts can be applied to distance-vector algorithms. In bandwidth-constrained and highly-connected networks (e.g., some wireless networks) distance vector's ability to summarize the data at each hop is an advantage because it limits the spread of topology updates, reducing bandwidth usage. As noted in section 2, our concern for robustness resulted in the use of link state in our initial research, but given FIRE's significant benefits in such network environments, we believe it would be valuable to explore adapting FIRE to support distance-vector algorithms.

Another important open problem is providing support for FIRE-style routing across multiple ASs. The main difficulty in inter-domain routing is developing a method to summarize the key features of an AS to its neighbors. Is there a single summary that meets all needs, or do summaries need to be constructed on a per-algorithm or per-property basis? Translating or interfacing between two different sets of algorithms and properties at border gateways presents another significant obstacle.

ACKNOWLEDGEMENTS

We are indebted to Robert Morris, Ram Ramanathan, and Robert Shirey for their invaluable comments on previous versions of this paper. Steve Kent provided expert guidance on several security issues. Benjie Chen assisted with the performance measurement of our prototype. Ayan Banerjee, Rahul Biswas, Matt Fredette, Fabrice Tchakountio, Laurie Thompson, and Greg Troxel contributed to the reference FIRE implementation.

REFERENCES

[1] C. Adams and S. Farrell, "Internet X.509 public key infrastructure certificate management protocols," RFC 2510 (Mar 1999).

[2] D. S. Alexander, W. Arbaugh, A. Keromytis, and J. Smith, "A secure active network environment architecture," *IEEE Network,* 12, 3, pp. 37-45 (May/Jun 1998).

[3] D. O. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "Requirements for traffic engineering over MPLS," RFC 2702 (Sep 1999).

[4] S. Bahk and M. El Zarki, "Dynamic multi-path routing and how it compares with other dynamic routing algorithms for high speed wide area network," *Proc. ACM SIGCOMM '92,* pp. 53-64 (Aug 1992).

[5] M. Bailey, B. Gopal, M. Pagels, L. Peterson, and P. Sarkar, "PathFinder: A Pattern-Based Packet Classifier," *Proc. USENIX OSDI '94,* pp. 115-123 (Nov 1994).

[6] A. Ballardie, P. Francis, and J. Crowcroft, "Core Based Trees (CBT): An architecture for scalable multicast routing," *Proc. ACM SIGCOMM '93,* pp. 85-95 (Sep 1993).

[7] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) -- Version 1 functional specification," RFC 2205 (Sep 1997).

[8] R. Callon, "Use of OSI IS-IS for routing TCP/IP and dual environments," RFC 1195 (Dec 1990).

[9] I. Castineyra, "Hop-Spec: specifying the capabilities of the hop within an internetwork," Technical Report 7813, BBN Technologies (1992).

[10] B. Chinoy, "Dynamics of internet routing information," *Proc. ACM SIGCOMM '93,* pp. 45-52 (Sep 1993).

[11] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Internetwork: Research and Experience,* 1, 1, pp. 3-26 (Jan 1990).

[12] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* 1, pp. 269-271 (1959).

[13] D. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," *Proc. ACM SIGCOMM '96,* pp. 53-59 (Aug 1996).

[14] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei, "Protocol Independent Multicast-Sparse Mode (PIM-SM)," RFC 2362 (Jun 1998).

[15] J. Gosling, B. Joy, and G. Steele, *The Java language specification,* Addison Wesley, Reading, MA (1996).

[16] M. Hicks, J. Moore, D. S. Alexander, C. Gunter, and S. Nettles, "PLANet: An active internetwork," *Proc. IEEE INFOCOM '99,* pp. 1124-1133 (Mar 1999).

[17] H. Holbrook and D. Cheriton, "IP Multicast Channels: EXPRESS Support for Large-scale Single Source Applications," *Proc. ACM SIGCOMM '99,* pp. 65-79 (Sep 1999).

[18] V. Jacobson, "Congestion avoidance and control," *Proc. ACM SIGCOMM '88,* pp. 314-329 (Aug 1988).

[19] S. Kent and R. Atkinson, "Security architecture for the Internet Protocol," RFC 2401 (Nov 1998).

[20] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)," Technical report, BBN Technologies.

[21] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)," RFC 2406 (Nov 1998).

[22] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE JSAC,* 14, 7, pp. 1280-1297 (Sep 1996).

[23] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," *Proc. USENIX Technical Conference,* pp. 259-269 (Jan 1993).

[24] J. McQuillan, I. Richer, and E. Rosen, "The new routing algorithm for the Arpanet," *IEEE Transactions on Communications,* 28, 5, pp. 711-719 (May 1980).

[25] J. Moy, "OSPF version 2," RFC 2328 (Apr 1998).

[26] S. Murphy, M. Badger, and B. Wellington, "OSPF with digital signatures," RFC 2154 (Jun 1997).

[27] G. Necula, "Proof-carrying code," *Proc. ACM POPL '97,* pp. 106-119 (Jan 1997).

[28] K. Nichols, V. Jacobson, and L. Zhang, "A two-bit differentiated services architecture for the internet," RFC 2638 (Jul 1999).

[29] R. Perlman, *Interconnections: Bridges and routers,* Addison Wesley, Reading, MA (Aug 1997).

[30] T. Rodeheffer and M. Schroeder, "Automatic Reconfiguration in Autonet," *Proc. ACM SOSP '91,* pp. 183-197 (Oct 1991).

[31] E. Rosen, "Vulnerabilities of network control protocols: An example," *CCR,* 11, 3, pp. 10-16 (Jul 1981).

[32] K. Sklower, "A tree-based routing table for Berkeley Unix," Technical report, UC Berkeley (1993).

[33] K. Sollins, "The TFTP protocol (revision 2)," RFC 1350 (Jul 1992).

[34] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM,* 46, 3, pp. 362-394 (May 1999).

[35] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," *Proc. ACM SIGCOMM '97,* pp. 25-36 (Sep 1997).

[36] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," *Proc. IEEE OPENARCH '98* (Apr 1998).

[37] R. White, A. Retana, and D. Slice, *EIGRP for IP,* Addison Wesley, Reading, MA (In press).