Argante2
Design Notes

James Kehl
ecks@optusnet.com.au

May 26, 2002

Chapter 1

Kernel Design

1.1 "Ring 0" - core essentials

1.1.1 Memory Access

```
typedef struct {
  union {
    unsigned long u;
    signed long s;
    a2float f;
  } val;
} anyval;
```

a2float is a floating-point type with sizeof(a2float) == sizeof(long) - float on 32bit systems, double on 64bit systems.

Within Argante, *all* memory is of this type, and accesses are always aligned to anyval block boundaries; this makes it easier to catch boundary violations, but makes string manipulation difficult and leads to 16 GB of addressable memory on 32bit systems (which have an 4 GB address space).

The central part of Argante is securing memory access - preventing code modification, call stack overwriting, heap corruption attacks, and so on (ad nauseum, sadly).

Currently the memory space is divided into 16384 blocks of 1 MB each. This has the sole advantage of simplicity. It's likely to change, as 1 MB is not a Goldilocks number - it's either way too small or way too large, but any changes will require changing the image file format.

```
#define A2_MEM_READ                1
#define A2_MEM_WRITE               2
#define A2_MEM_MAPPED              4

struct memblk {
  unsigned int mode; /* Access permissions */
  anyval* memory; /* Real pointer */
  unsigned int size; /* Size */
  int destroy_scnum; /* Syscallnum to do a basic FREE */
  unsigned alib_id; /* so we can get rid of lib's mem */
};
```

Memory pages have the type shown above, though they should not be accessed directly. MAPPED memory is accessible like normal memory, but cannot be resized or have its permissions changed. Freeing it will call the syscall given in 'destroy_scnum' if it is nonzero.

size is in units of anyval.

alib_id is provided to allow the dynamic linker to free pages that were loaded from an image file which was subsequently removed. Pages allocated at runtime will not be recognized.

```
unsigned mem_alloc(struct vcpu *curr_cpu, unsigned size,
        unsigned flags);
unsigned mem_realloc(struct vcpu *curr_cpu, unsigned addr,
        unsigned newsize);
void mem_changeperm(struct vcpu *curr_cpu, unsigned addr,
        unsigned newflags);
void mem_dealloc(struct vcpu *curr_cpu, unsigned addr);
```

The above functions will allocate, resize, free or change the permissions of a block of memory, or throw an exception if they fail. When they return an unsigned value, it is an address from Argante-space, i.e. usable in mem_ro or mem_rw...

```
const anyval *mem_ro(struct vcpu *curr_cpu, unsigned addr);
anyval *mem_rw(struct vcpu *curr_cpu, unsigned addr);
```

The above functions permit access to a single anyval for readonly or read-write usage. This is faster than paired get_mem_value/set_mem_value calls, but kernel code must not override the 'const' and write to memory that was checked out as readonly. Use _rw, this checks for write permission.

```
#define dwords_of_bytes(a) \
        ((a / sizeof(anyval)) + ((a % sizeof(anyval)) ? 1 : 0))

const anyval *mem_ro_block(struct vcpu *curr_cpu, unsigned addr,
        unsigned dwords);
anyval *mem_rw_block(struct vcpu *curr_cpu, unsigned addr,
        unsigned dwords);
```

The above functions the permissions and validity of a large block of memory at once. It is possible to cast the return value to a char * or const char * if needed.

dwords_of_bytes gives the number of anyval units required for 'a' bytes, and is useful for calculating the 'dwords' arguments of the _block functions.

```
int kerntoa_strcpy(struct vcpu *curr_cpu, unsigned addrto,
        int size, const char *from);
int kerntoa_memcpy(struct vcpu *curr_cpu, unsigned addrto,
        const char *from, int size);
int atokern_memcpy(struct vcpu *curr_cpu, char *to,
        unsigned addrfrom, int size);
```

kerntoa_strcpy copies a null-terminated string into Argante-space. The rest are similar and should be self-explanatory.

1.1.2 Opcode Design

Argante has a fixed bytecode size. Without this, it is impossible to really know what the code does. On other architectures, one perfectly ordinary-looking instruction can become another if you jump into the middle of it, and that also alters every instruction executed after that.

```
struct bcode_op {
  unsigned char bcode;
  unsigned char type;
  short reserved; /* 32-align arguments for speed */
  anyval a1;
  anyval a2;
};
```

Consider an instruction like ADD. Logically, it should work for signed, unsigned and floating-point types, and each combination needs a different code to implement.

Also, you should be able to use a constant, a register or the contents of a memory address, but that doesn't change the actual instruction, it only changes the location of the arguments in real memory, and you still need to know the type of the argument.

```
#define TYPE_UNSIGNED 000
#define TYPE_SIGNED 001
#define TYPE_FLOAT 002

#define TYPE_IMMEDIATE 000
#define TYPE_REGISTER 004
#define TYPE_POINTER 010

#define TYPE_A1(a) ((a) << 0)
#define TYPE_A2(a) ((a) << 4)
#define TYPE_VALMASK (TYPE_UNSIGNED | TYPE_SIGNED | TYPE_FLOAT)
```

The first 2 bits of a bcode_op's 'type' field are used for the type of argument 1, the next 2 are flags designating immediate values, registers, pointers, or pointers within registers. The next 4 are like the last 4, only for argument 2.

The type bits are used to find what instruction to call. The flags specify how

to get to the arguments - in the case of immediates, it's just the address of the anyval within the bcode_op structure.

To stop people overwriting immediates, the instructions which change an argument need to be known about. When an image file is loaded, validate_bcode_page from imageman.c is run. This checks that each referenced instruction actually exists, that it doesn't try and overwrite immeduates, that it doesn't use register 33, and finally precalculates the offsets within the JIT table that each instruction will require (hey, I could remove the addition too! **FIXME**)

The precalculation's for speed, not security, but it does prevent anything that changes an opcode from having any effect.

There's a script that does all the hard work of generating the JIT table and tracking which arguments of which instructions are readonly and read-write. It gets its information from lines like:

```
/*! MOV 2 u RW u RO = cmd_mov_uu */
```

The above lines merely says that *all* MOV instructions have 2 arguments, the first of which is written to and the second of which is readonly; and that cmd_mov_uu is the name of the function which handles 2 unsigned arguments.

It really should have an opcode number in there, too. **FIXME**

cmd_mov_uu, written fully, is just:

```
static void cmd_mov_uu (struct vcpu *curr_cpu, anyval *a1,
  anyval *a2) {
  a1->val.u=a2->val.u;
}
```

Syscall2 - the rationale

As instructions are all of equal size, a one argument operation takes the same space as a two argument operation - the one argument op is wasting 4 bytes.

In the case of syscalls, which can require any number of arguments, and take them in r0 - r30, the problem gets worse - it's 4 bytes wasted in the 1-argument syscall instruction itself, and another 12 on a MOV instruction to pass another argument to the syscall.

Equally, when debugging asm using IO_PUTINT, you have to back up r0, move the number to print into r0, perform the syscall, and restore r0. And hope you didn't clobber something when backing up r0 in the first place. Annoying, ugly, and a potential cause of errors.

So, the new SYSCALL2 instruction takes two arguments, and passes its second argument to the syscall function.

This is nice when writing lots of asm, but high level languages will find it easier to ignore, and it's not a very significant optimization when it comes to large, multiple-argument syscalls. It's also argued that it makes code analysis harder, though the assembler warning when a non-syscall2 function is syscall2'd doesn't leave much confusion for me.

Perhaps syscall2 will be "voted off the island". Stay tuned.

Future Changes

The following opcodes were badly thought out and should change in some future version. This will probably happen at the same time as adding a 'address' field to data in the image format (and numerous other fixes).

- alloc (takes u:size, u:perms, writes address to size) is badly thought out. Who wants freshly-created read only memory? It should be more like realloc (addr, size).
- realloc (takes u:addr, s:perms) should be renamed. This is the change-permissions opcode and will never change its arguments, unlike realloc (u:addr, u:size).
- setstack will also be improved. There is currently no way to set the pointer, and resizing could be made cleaner. More thought required on this.

1.1.3 Exceptions

So many programmers are too lazy to check return values. And in most cases, they don't really care, because if one call fails the rest of the function can't keep going.

Hence the rise of exceptions. But as it's hard enough checking return values, running CHECK_FAILURE_FN when it's required is damn near impossible. So throw_except is implemented with longjmp, and you have to be able to deal with not having your function finish properly. Don't acquire a lock or malloc() a block you might not be able to release. alloca() is better.

Subexception handlers may be possible if this causes real problems.

Also, exceptions can only be thrown from within a running VCPU. This means functions which are used outside this context - say, image_load, or the vcpu_start or vcpu_stop functions of a module - can't throw exceptions.
(Note: we could allow exceptions in those functions. We'd just need a handler!)

1.1.4 Multithreading

When you are running VCPUs and interpreting keyboard input all from one thread, it becomes painfully difficult to avoid blocking calls. Under Unix, some functions are not available in nonblocking varieties (unless you want to rewrite them, that is), like readline() or gethostbyname().

So, you can't do it all from one thread.

One burden this places on you is to make sure you don't cross threading lines. Writing to static variables from VCPU code is out, as is poking at other images. Poking at running VCPUs from manager (main-thread) code is also out. For most big modifications (module loads/unloads) the VCPUs have to be spun down.

The other burden is that to spin down a VCPU means anything more specific about where a process is apart from curr_cpu->IP is lost. There should either be only one cancellation point per syscall (so the operation is atomic: done or not done) or a register should be modified to track the process. Otherwise we will 'stutter' - very untraceably.

1.2 "Ring 1" - syscalls, modules, and userspace API

1.2.1 Modules

To keep the non-optional (and therefore most security-critical, for there is no way to change the statically linked code barring a recompile) parts of Argante as small (and understandable/reviewable) as possible, the system interactions are partitioned up into modules. On most systems, these are dynamically loadable; on the rest, they can still be changed with ./configure options.

From the stats, just as many people download the Windows, non-dl binaries as the source, so static linking is needed as a fallback...

Four functions are needed in every module:

- `int module_init(unsigned lid)` is called when the module is loaded. Executed in manager context, so is guaranteed to be the only running part of the module: so setting globals is OK.

  lid is 'library id', a unique identifier for your library which acts as a key into the reserved structure and FD tables. Store this in a static global if you need it later.

  Return 0 for success and 1 for failure - if 1, module is unloaded and syscalls stay unavailable.

- `void module_shutdown(void)` is called when the module is unloaded. Again, this is the only part of module running when this is called.

- `void module_vcpu_start(struct vcpu *vcpu)` is called when a new VCPU starts. Other VCPUs may be using this module at this stage, and exceptions are not allowed. (No code has been executed and so no handlers can be set.)

- `void module_vcpu_stop(struct vcpu *vcpu)` is called when a VCPU stops. Again, no exceptions allowed - this is already a dead CPU, and that could revive it. Bad.

These hooks have proved sufficient, so far. Anything requiring more details should probably be written into the core kernel.

1.2.2 Reserved Structures

Due to multithreading requirements, you cannot use static/global variables to keep data about a particular VCPU. So, to store per-VCPU data, we have the 'reserved structure' array, which provides one void pointer per VCPU per module. You can store whatever you like in it; a malloc()d structure being most appropriate here.

The library ID passed to module_init is needed to use these functions. Most of the time you'll only use set_reserved in vcpu_start. Be sure to free in vcpu_stop!

```
void *module_get_reserved(struct vcpu *cpu, int lid);
int module_set_reserved(struct vcpu *cpu, int lid,
  void *newdata);
```

set_reserved returns 1 if it fails, so you can deallocate the data and assume crash position. One possible cause of failure is a corrupt LID. Another is that the system is out of memory.

get_reserved may return NULL if set_reserved hasn't been called or it failed. If it returns NULL, your best bet is to throw an exception (I recommend OOM). The alternative to aborting is to dereference a NULL pointer and bring the system down, so always check the return value!

1.2.3 Syscalls

```
#define SYSCALL_ARGS struct vcpu *curr_cpu, const anyval *arg
typedef void syscallfunc (SYSCALL_ARGS);

extern int register_syscall(unsigned id, syscallfunc *f);
extern int unregister_syscall(unsigned id);
```

register_syscall and unregister_syscall should be called during module_init and module_shutdown (and ONLY then). These create and destroy the association between a numeric syscall ID and the function it should be dispatched to. This is stored in a hash table.

The function takes the VCPU pointer (of course) and arg - which is a *readonly* argument - either r0 if the function's been syscalled, or the second argument of a syscall2.

1.2.4 Autogenerator

As with the JIT table, syscall functions are so - well, homogenous - that it's worth automating the syscall registration/deregistration and maintainance of the compiler's ID→syscall tables.

To do this you'll need to be writing in C (I don't know ADA, nor do I know anyone else who'll admit to knowing it :P). Using comment-bang lines (like for the JIT) a perl script will do the rest, complete with #ifdefs so that static linking works.

```
static inline int module_internal_init(int lid):
static inline void module_internal_vcpu_start(struct vcpu *cpu):
static inline void module_internal_vcpu_stop(struct vcpu *cpu):
static inline void module_internal_shutdown(void);
```

These should be static at the very least, if not static inline. They are *internal* and should not be available externally, and most especially not to other modules which might be linked with yours.

To include the autogenerated code which the kernel will be interfacing with, write

```
#include "file_name.h"
```

where file_name is the name of your module. Any existing header of that name will be overwritten. (I suppose the file name could have been `file_name.hgen` ...)

1. `/*! allowed x - y */`
2. `/*! NEW_CALL1 = new_call1 */`
3. `/*! NEW_CALL1 99900 = new_call1 */`
4. `/*! NEW_CALL1 SYS2 = new_call1 */`
5. `/*! NEW_CALL1 99900 SYS2 = new_call1 */`

The first line tells the autogenerator that this module has been assigned a range of syscall numbers. The range 99900 - 99999 is defined as a testing range: feel free to use this before you've been assigned a range, but if you release code that uses it, Sendmail ("the daemon from hell") will come round and have a little chat with you.

The second line tells the autogenerator that the syscall named NEW_CALL1 is implemented in new_call1, and that the autogenerator should pick a number for it. This number should be written into the code ASAP. Think what happens if FS_RENAME's number became that of FS_DELETE's.

The third line is the same as the second, only with the number specified. This is good.

The next two lines are the same as the previous two, only they say that new_call1 uses its arg parameter and so is a SYSCALL2.

1.2.5 Heirarchical Access Control

I hope everyone understands ACLs (access control lists) and inherited permissions. (If not, you'd better ask the guy who's r00ting your box right now.)

To control access to resources, Argante uses a method called Heirarhical Access Control. This defines what operations may be performed on what resources (be they files, sockets, or system statistics). These resources make up a filesystem-like tree. The operations, too, are defined to form a tree: so we have /open/, /open/read/, /open/write/, and /open/write/overwrite.

For files, the resource name is NOT the filename. There needs to be a 'namespace' prefix: /fs/etc/hosts is the file /etc/hosts, for example. /tcp4/192.168.0.1 is an IPv4 host, very distinct from /fs/192.168.0.1, which is a (yawn) file.

So that every permission does not have to be specified for every file, inheritance is used. If at a particular level, a permission is unspecified, it is inherited from a more general level. (The operations tree is searched before the resource tree.)

Argante2's HAC is a little different from Argante1's. The HAC is actually stored in a heirarchical structure (actually doubly nested hashed linklists, though someday it will change to something less memory hungry) rather than being implemented as string comparisons.

This means you can specify HAC rules in any order whatsoever, and you can't specify wildcards: the generalization of rules has to be done during the module design phase. Entries like /file/create and /file/delete should be avoided in favour of /create/file and /create/directory - assuming you will grant permissions for directory/file creation more ofen than file creation/deletion...

Remember - every 'directory' is a new layer, consuming memory, and taking

time to traverse. Don't be unnecessarily specific: do you really need all the details in /fs/fops/local/create/file/regular? /fs/ should be in the resource path, to start with.

Argante2's HAC DOES NOT check for wildcards and directory traversals. This is not its job. The filenames '*' and '..lck' are valid filenames, and in namespaces other than the filesystem '..' itself might be safe and '#!' dangerous. To cut out directory traversals, the function fold() is provided. (Don't stick the namespace prefix on before this, or someone can jump namespaces.)

`VALIDATE(dir, atype)` is the usual way to check HAC permissions; this macro requires curr_cpu to be in a variable called 'curr_cpu', of all things. If your syscalls stick to SYSCALL_ARGS then you're set.

If VALIDATE fails, it throws an exception. So don't malloc() or open a file or do anything that might not get cleaned up before you call it. alloca() is preferred over malloc() for just this reason. Beware, though, of alloca()'ing large chunks of memory; it fails disasterously (Chernobyl-style).

If you can't use VALIDATE, `validate_access(curr_cpu, dir, atype)` is the underlying call. It returns nonzero on access failure.

1.2.6 Virtual File Descriptors

This is designed to be used for, like the name suggests, virtual file handles. The VFD facilities have significant advantages over reserved structures for this purpose, and have their own limitations which aren't significant for this use.

One very significant motivation for using VFDs is that you don't need to mess with lots of resizing of reserved structures. The limitation is that each VCPU has an upper limit on virtual file descriptors, which are shared between all modules. (Of course, this is an intentional, adjustable limit.)

```
int vfd_alloc_new(struct vcpu *curr_cpu, int lid);
```

This returns a unique number which identifies your new VFD. Your OPEN call is pretty useless if it doesn't return this to the user.

```
void *vfd_get_data(struct vcpu *curr_cpu, int lid, int handle);
void vfd_set_data(struct vcpu *curr_cpu, int lid, int handle,
  void *newd);
void vfd_dealloc(struct vcpu *curr_cpu, int lid, int handle);
```

These all throw a ERR_BAD_FD exception if passed a handle that wasn't created by your library (according to lid).

```
int vfd_find_mine(struct vcpu *curr_cpu, int lid);
```

vfd_find_mine is worth noting; it returns a handle if your module owns a VFD. It's most useful for destroying all your VFD's in a vcpu_stop routine - mind, though, that you actually vfd_dealloc or vfd_find_mine will keep returning the same number...

Despite the name, these aren't limited to IO. You could write a MySQL module and use VFDs for database connections, or maybe even queries. You might write a hashtable module and use VFDs for particular hashtables. Anything that a VCPU might want multiples of is a candidate.

There is also a common layer for VFDs that DO stick to the I/O paradigm, that allows CFD_WRITE to write to files or consoles or sockets or syslog or... depending on what module created the VFD, and also allows modules to send/read kernel-space data via VFDs - so PUT_HEX does not get implemented 20 times...

1.2.7 Common Operations layer

```
typedef void cfdop_close_fd (struct vcpu *curr_cpu, void *vfd);

/* for agents to create VFDs. */
typedef int cfdop_create_fd (struct vcpu *curr_cpu,
  const char *desc, int in, int out);

/* returns "Block size" - maximum size to read/write at once. */
typedef int cfdop_start (struct vcpu *curr_cpu, void *vfd);

/* returns bytes read/written */
typedef int cfdop_write_block (struct vcpu *curr_cpu, void *vfd,
  const char *buf, int size);
typedef int cfdop_read_block (struct vcpu *curr_cpu, void *vfd,
  char *buf, int size);

/* CFD operations table version 1. */
struct cfdop_1 {
  cfdop_start       *read_start;
  cfdop_start       *write_start;
  cfdop_read_block  *read_block;
  cfdop_write_block *write_block;
  cfdop_close_fd    *fd_close;
  /* A unique endian-independant code (ie a string) for
     accepting agent VFD's. Only used for fd_create. */
  int fd_desc;
  cfdop_create_fd   *fd_create;
};
```

To implement CFD, you put some function addresses into this table (If your VFDs are always RO, then you might leave some fields NULL. Very few modules implement agent-FDs, and most leave fd_create NULL.

The way it works is: say CFD_READ is called. It looks up what module that VFD came from, and finds the table that's associated with that module. From that table, read_start is called with the VFD data. It might, say, check the HAC. Then it returns the block size writes should occur in (mostly for historical reasons). The block size MUST be a multiple of sizeof(anyval). Then read_block is called repeatedly until all the data is read.

The agent-fd calls are to allow management agents to create VFDs for consoles, or GUI connections, through a module which can speak the protocol (in this case, VT100 or X). The fd_create call will soon change to accept flexsocks instead of file descriptors for portability.

The following calls allow searching and registering tables. _lid_set should be called during module_init.

```
/* get/set a table by lid */
extern void cfdop1_lid_set(unsigned lid,
  const struct cfdop_1 *a);
extern const struct cfdop_1 *cfdop1_lid_get(unsigned lid);
/* get a table by a fd_desc - for agent-fds */
extern const struct cfdop_1 *cfdop1_fddesc_get(int fddesc);
/* Get the table for a vfd */
extern const struct cfdop_1 *cfdop1_vfd_get(struct vcpu
  *curr_cpu, unsigned handle);
```

1.2.8 "Alib": dynamic linking

This should be avoided in favour of IPC. However, there is no IPC functionality yet.

1.3 "Ring 2" - management

1.3.1 Agents

TBA. It's safe to say anyone who's ever used the A2 console knows there needs to be a better way...

1.3.2 FlexSock

For a module to talk via an agent-FD, it has to have some way of writing and reading data with that FD. But is it a file handle, a FILE *, a Win32 named pipe, or an OpenSSL connection, each of which have their own write method?

Hence - the FlexSock abstraction layer to isolate all the messy details. OpenSSL has something similar in its BIO functionality - wouldn't it be nice if everyone had OpenSSL!

Details, as usual, in flux.

1.3.3 Remote IPC - DRAFT

Routing

Nobody has written a single line of rIPC code for this version yet, so this is just ideas... forgive the lack of ASCIIs.

In *any* network, you have two types of element - the hub and the node. A node cannot connect directly to another node, nor can it connect to more than one hub. If it connected to multiple hubs, it would be represented as a node connected to a hub connected to the other hubs.

In A2 rIPC, a node's only function is computation, and a hub's only function is routing and communication. The only overlapping functionality is in the common network protocols, the lower levels of which should be encapsulated within FlexSock. Node→hub, hub→node and hub→hub protocols will not have much in common.

All comms are in network byte order (big-endian).

When a new node connects to a hub, all other hubs need to know how to send messages to it (the other nodes just depend on their hub for this). The new node's hub sends a ADDED message to all the hubs it is connected to.

```
struct msg_ADDED {
  uint16_t node_address;
  uint16_t hops;
  uint16_t round_trip_time;
}
```

Node addresses are 16 bits for forced obsolescence. If the address space is ever exhausted within a single rIPC network (!), it will be a lot less problematic to extend at that stage than an exhausted 32 bit address space. As a hub will never be an endpoint for a message, they are not rIPC-addressable.

When a hub recieves an ADDED message, it adds one to 'hops', increases the 'rtt' by the rtt of the sender, and files the new data and sender under the node address. If the sender already advertised this node, the previous data is replaced.

A hub MUST not accept a direct connection using a node address which the hub already has a path for (especially a direct, hops=0 one, though the hub SHOULD check if the original connection died). The only time this will happen is when two nodes use the same address, which is an accident or an attack.

If the new data is 'better' in terms of rtt than the last 'best' for that address, the new data is sent out to all hubs (except the sender). Hubs MAY delay (quench) this retransmission if the data is changing quickly.

Hubs MAY ignore ADDEDs that exceed their limits on hops or rtt. If so, these limits should be chosen carefully to avoid "one-way streets". Hubs MAY also forget the 'worst' records if they have too many, although they will never have more than one per hub they are connected to, and MUST retain at least two records. Hubs MAY allow a 'fuzz factor' in order to recycle common data.

When a hub connects to an existing network, it MUST broadcast ADDEDs for all the nodes already in its table. This is to allow existing networks to be joined.

```
struct msg_DROPPED {
  uint16_t node_address;
}
```

When a node disconnects from its hub, the hub broadcasts a DROPPED message. When a connection between hubs is lost, every path using the lost hub must be removed. Any nodes which the hub no longer has a path for then generate a DROPPED message, which prompts other hubs to forget that path, and rebroadcast the DROPPED message if they have no path to that hub. Quenching MUST NOT occur here.

So, when a hub gets a DROPPED message, it removes the entry for the source from the node's paths, and if the hub has no more paths for that node, it tells all the other hubs that it cannot find that node with more DROPPED messages.

If a node's best path has been lost, but the hub has a path remaining, it MUST send all hubs it knows (even the one which sent the DROPPED), an ADDED

message containing the new data. Otherwise the fastest, or even only, route might be overlooked.

```
struct msg_DATA {
  uint16_t dst_address;
  uint16_t src_address;
  uint16_t hops;
  char data[];
}
```

When a hub recieves a data packet, it accepts a sacred responsibility to get that packet to someone else. First, it finds the server with the lowest listed rtt for dst_address, and attempts to hand the message off. If that fails, the hub→hub connection is considered dead, and DROPPEDs generated. And then the hub tries the next fastest route.

If the hub ends up running out of routes, the DATA becomes a DATA_REJECT, src and dst addresses are switched, hops are set to zero, and the data is dropped. If a DATA_REJECT is rejected, then a warning should be generated and the packet MUST be dropped.

Routing - Caveats

It is impossible for a rogue node to fiddle the routes, and a node attempting a DoS by adding and removing itself can be catered for by quenching - introducing a time lag into ADDED rebroadcasting.

However, a rogue hub could easily steal data and cause DoS. The only solutions are configuring each hub to only accept certain other hubs, or to use authentication. Either solution relies on trust. Hub→hub connections should use SSL in any case. Node→hub connections are likely to be within computers and secure networks, so SSL is mostly overkill. (But flexibility never hurts).