

# High noon at the OKE Corral: Code Organisation and Reconfiguration at Runtime using Active Linking

(Extended version of the paper on the same subject published at IWAN'2002)

Herbert Bos and Bart Samwel

Leiden Institute of Advanced Computer Science (LIACS), Leiden University, The Netherlands

Email: {herbertb,bsamwel}@liacs.nl

Phone: +31-71-527 7033, Fax: +31-71-527 6985

**Abstract**—This paper describes the OKE Corral, an active network environment which allows third-party active code to control the code organisation at any level of the network node. This includes the kernel. The underlying code is structured much like components in a 'Click'-router that may be connected or disconnected at runtime. Using this, active packets are able to reconfigure predefined components in the networking code, regardless of their location. Moreover, using the safe programming model of the open kernel environment, they are able to load and link their own components at any place in the datapath and at any level in the packet processing hierarchy.

**Index Terms**—open kernel environment, active networks

## I. INTRODUCTION

For reasons of safety, implementations of active networks (ANs) tend to sandbox active code in user space, either locally or at some remote processing node. Moreover, the code is often interpreted, which slows down the performance of the active code considerably. Even in non-active environments interpreters are frequently used whenever application-specific code is loaded in the kernel. A well-known example is packet filtering in BPF, where the packet filters consist of expressions in a stack-based language that is interpreted in the kernel.

In previous work, however, we have shown how the open kernel environment (*OKE*) allows fully optimised native code to be loaded in a Linux kernel by parties other than `root` in a safe manner [BS02]. The *OKE* provides a safe, resource controlled programming environment: code can be restricted in stack, heap and CPU usage, as well as in the access to kernel functions and memory. The amount of restriction depends on the privileges given to the code-loading party in the form of credentials. Sample applications in the field of packet transcoding showed that

carefully-written OKE code outperforms implementations that filter the packets in the kernel, while processing them in userspace.

In this paper, we describe our progress in the implementation of the *OKE Corral* (Code Organisation and Reconfiguration at Runtime using Active Linking), an environment for building high-speed active networks, that allows fully optimised native code to be loaded and configured anywhere in the processing hierarchy, including the kernel. The contribution of this work is that three novel technologies in the field of networking and open systems (open kernels, the 'Click software router' model and active networks) are combined to provide a platform for fast programmable packet processing with explicit separation between control and data flow. In the *OKE Corral* high-speed packet processing can be conveniently managed by slow-speed control code. The essence can be summed up as follows:

- 1) We borrow the LEGO-like software organisation model that was advocated by the 'Click' router project [CM01] both to build fast data-paths and to implement paths for control traffic.
- 2) One of the components on the control path is an AN runtime (if required, this path *can* also be used for data packets, but because the control path is fairly slow, this is much less efficient).
- 3) The configuration and implementation of the other control and data path components can be initiated by active packets, remote parties, or both.
- 4) The *OKE* is used to ensure that kernel-level implementations of the path components are safe.

Although the individual components may not be new, to the best of our knowledge there does not exist any system that provides the following combination of features in a commonly used operating system: (a) programmability of both kernel and user space with fully optimised

native code, (b) while still providing full resource control and safety with respect to memory, CPU, available API, etc., and (c) allowing for flexibility in the amount of programmability permitted on a node, and (d) where control over fast native code components is exercised by slow-speed active applications (AAs), (e) by means of a simple 'Click-like' programming model,

The *OKE Corral* described in this paper has been fully implemented and a prototype implementation has been evaluated (see Section IV). We stress, however, that not all issues concerning the use of the *OKE* for active networking are addressed in this paper. In particular, we have not considered the question of heterogeneity (shipping fully compiled and optimised code in a highly heterogeneous environment), or the question of scaling the trust relationships to networks as large as the Internet (as the *OKE* relies on trusted compilers, the issue of whether and under what circumstances to trust compilers in a remote domain is non-trivial). A possible solution to both problems might be to ship the code in source format and trust only local compilers.

The remainder of this paper is organised as follows. The *OKE Corral* architecture is discussed in Section II, and the prototype implementation of the architecture in Section III. The prototype is evaluated in Section IV. Related work is discussed in Section V, and conclusions are drawn in Section VI.

## II. ARCHITECTURE

As illustrated in Figure 1, the *OKE Corral* builds on three technologies: (1) the open kernel environment, (2) one or more AN runtimes, and (3) packet channels that implement control and data paths and link the various components (processing elements and queues) seen by the packets as they traverse the network node. As indicated in the figure the black boxes represent processing elements and queues on the packets' data-path. The various boxes and functions in Figure 1, as well as the way in which they interact will be explained below. To the right of the architecture we have indicated the approximate mapping of *OKE Corral* components on the DARPA reference architecture of an active node. By necessity this is only an approximation. For instance, depending on the configuration the kernel may or may not be dynamically programmed (i.e. run AAs in its execution environment).

### A. Corral terminology

Before discussing the architecture in detail, we first need to introduce some of the terminology. The terminology is intuitive and coincides to a large extent with that of the Click-router project. However, there are some differences.

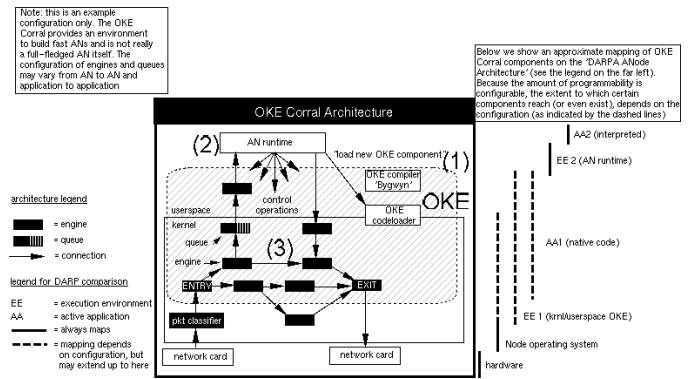


Fig. 1. Overview of the *OKE Corral*

Processing elements are called "engines" in the *OKE Corral* and may have multiple input and output ports that may be logically attached to other elements to form connections (drawn as arrows in Figure 1). A data transfer whereby the source element takes the initiative is called a *push* operation, while a transfer initiated by the destination is called a *pull* operation. Connections are always either of the *push* or the *pull* type. Engines are normally also either 'push' or 'pull', but a hybrid form, known as *pull2push* is also possible. A *pull2push* engine pulls data from a source element and pushes it to a destination element, and hence changes the pull into a push.

The inverse of a *pull2push* element is a queue. A queue accepts pushes (of data items) on its input, and pulls on its output, and thus may be termed a *push2pull* element. Queues and *pull2push* elements together are called *latches*. Queues may be filled and emptied by more than one engine. As shown in Figure 1, engines and queues are connected and disconnected via control operations.

The path followed by a particular packet is known as a "channel". In Figure 1, for example, the path formed by the entry engine, the two boxes on the right of the entry engine, and the exit engine is a channel. Channels may consist of "subchannels", which start at an engine and end either at a latch, or at the packet exit engine. For example, the channel from the packet entry engine via the execution environment to the packet exit engine consists of at least the following two subchannels: (1) from entry to queue, and (2) from the engine in user space all the way to the exit engine. Subchannels are either *push* or *pull*.

In contrast to the Click approach, queues and engines may reside in the kernel, in user space, or even on remote machines. Wherever they reside is known as the queue or engine's "domain". Similarly, they may exist either inside the *OKE* (in which case they are subject to checks and resource limits), or as native, unprotected code. Even the execution environment (EE) which is shown outside the

*OKE* box could easily be moved inside the *OKE*. In other words, Figure 1 only serves as a high-level overview of one out of many possible configurations.

The packet classifier in the figure determines which packets will be relegated to the AN's channels. It is really part of the *OKE* environment setup code (ESC) for the AN, but we will see that it lies beyond the reach of the AN and this is why it is drawn outside of the *OKE*'s box.

### B. *OKE*, AN runtime and Channel interaction

When an AN runtime is instantiated, it is initially provided with a channel consisting of two engines: the packet entry engine and the packet exit engine. All the AN's packets are first pushed to the entry engine, which automatically leads to a push to the exit engine, from where they are transmitted onto the network. Each of the pushes is implemented as a function call, and is executed immediately and in the same thread of control.

The AN is allowed to disconnect the two engines, reconnect them, or connect them to new components inserted between these engines, all at runtime. For example, a trivial AN implementation might take the following steps to receive all packets in its runtime: (1) disconnect the two engines, (2) reconnect the entry engine to queue (which is a standard component in the *OKE Corral*, which can be used as much as the AN's resource limits allow), (3) implement an engine's interface for the runtime (essentially making the runtime a userspace domain engine itself which 'pulls' packets from the queue and pushes them up into the runtime), and (4) implement the runtime's send operation as a push to the exit engine. From that moment onward, all incoming packets classified as AN traffic are automatically pushed onto the queue, and from there pulled up into userspace.

The AN is given a set of standard components (engines and queues) with which to build channels (subject to the privileges given to the AN). Depending on their tasks, these standard components can be highly optimised and may incur few (if any) checks at runtime. Besides such standard components, the AN is able to load entirely *new* components at any level of the processing hierarchy, including the kernel. In case the component is a bit of code in the runtime, we may rely on the runtime (for example, a Java Virtual Machine) sandboxing it. For native code to be loaded in the kernel, however, this is certainly not the case and for such code the *OKE* is needed. The *OKE* is able to restrict code according to the code-loading party's privileges. Privileges are communicated in the form of credentials. In *OKE* terminology, a client's credentials determine its *role*. In this way, a highly untrusted party may be allowed to load code with very few privileges and

many dynamic checks, while a highly trusted party (e.g. the system administrator) may benefit from a much more relaxed security policy. The way this is implemented in the *OKE* will be discussed in more detail in Section III-A.

### C. Control and data channels

Using the above techniques, an AN is able to build fast channels where processing is done in optimised native code and where the next processing stage is always just a function call away. At the same time we also use channels to implement slow-speed control paths which commonly lead to AN runtimes in user space (or even remote hosts) and which are used to carry the active packets. The active packets carry the control code. Given the appropriate privileges they are able to replumb, or add new elements to, the data-path. Using *OKE* and privileges in the form of credentials, the amount of programmability that is allowed on the data-path is configurable, which is useful if active networks are to scale to the size of the Internet.

## III. IMPLEMENTATION

In the following three subsections, we will discuss in detail the main components that make up the *OKE Corral*.

### A. The Open Kernel Environment

Allowing third-party code into the kernel normally jeopardises security constraints as the code can 'do anything'. From a performance perspective, on the other hand, it would be useful. In the *OKE* instead of asking whether or not a party may load code in the kernel we ask: what is such code allowed to do there? *Trust management* is used to determine the privileges of user and code, both at compile time and at load time. Based on these privileges a *trusted compiler* may enforce extra constraints on the code (over and above those imposed by the normal language rules). As a result, the generated code, once loaded, may incur dynamic checks for safety properties that cannot be checked statically.

In this paper we only give a high-level overview of the *OKE*. A detailed explanation was presented in in [BS02]. A pre-release of the *OKE* will be made available from [www.liacs.nl/~herbertb/projects/oke/](http://www.liacs.nl/~herbertb/projects/oke/). For the present discussion, two components of the *OKE* are essential: (1) the code loader, which loads a user's code in the kernel, and (2) the *byg-wyn* compiler, which compiles user's code according to the rules corresponding to the user's privileges. Both are also shown in Figure 1.

1) *Code loader*: The existing Linux code loading facilities have been extended with a new code loader (CL) which accepts object code, together with authentication and credentials, from any party. Anyone with the right credentials for the code is allowed to load it into the kernel, so there is an (implicit) record of who is authorised to load what modules. The CL checks the credentials against the code and the security policy and loads the code if they match. The process is illustrated in Figure 2.

The trust scheme and authorisation checks are implemented using KeyNote [BFIK99] and the OpenSSL library. Trust may be delegated. At start-up time, the CL loads a security policy, which contains the public keys of the clients that are permitted to load kernel modules. The CL and ‘trusted’ clients are then able to delegate trust to other clients. Trust delegations are encoded in credentials containing the public keys of both the authoriser and the licensee, as well as the ‘rights’ granted by the authoriser to the licensee. For example, an authoriser may grant the right to ‘load a module of *type X* or *type Y*, but only under *condition Z*’. A ‘type’ here denotes the privileges given to the code, e.g., to access certain kernel data structures, to use a certain number of cycles and a certain amount of memory, etc. The ‘condition’ may contain environment-specific stipulations, e.g., that the loading rights are only valid during office hours. A module type is instantiated when source code corresponding to the type is compiled. The trusted compiler generates an unforgeable ‘compilation record’ which proves that module *M* (identified by its MD5) was compiled as type *T* by this compiler.

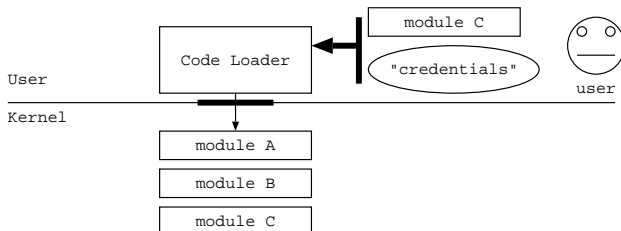


Fig. 2. User loads module in the kernel

2) *Elastic languages and APIs*: When loading third-party code in the kernel, problems arise when it is allowed to follow arbitrary pointers, call arbitrary functions, use unrestricted amounts of CPU time, etc. It is crucial that we guard against malicious or buggy code. What we have tried to avoid, however, is the definition of yet another safe language which is only useful for implementing filters, say, and/or runs inside an interpreted environment. The problem with such languages is that they necessarily restrict towards the lowest common denominator, while we would like to have a single language that is automatically restricted on the basis of explicit privileges. A single lan-

guage is preferable to many special-purpose languages for many reasons, e.g., consistency, learnability, maintainability, flexibility (new requirements can be catered to by the same language), etc. Moreover, the interaction with the rest of the kernel is an issue. All users benefit from using a language like C to facilitate the interfacing of their code to the rest of the kernel.

We therefore allowed a C-like programming language to be restricted in such a way that, depending on the client’s privileges more or less access is given to resources, APIs and data (and/or more or less runtime overhead is incurred). As C itself is not safe and the possibilities of crashing or corrupting a kernel using C are endless, we opted for *Cyclone*, a crash-free language derived from C which ensures safe use of pointers and arrays, offers fast, region-based memory protection, and inserts few runtime checks [JMG<sup>+</sup>02]. However, for true safety and speed, we needed both more and less than what was offered by Cyclone. For example, safe usage of dynamically allocated memory in Cyclone depends on the use of a garbage collector, which we had to reimplement completely to make it work in a Linux kernel. Many of the really hard problems (such as resource limitation, module termination, and the sharing of memory/pointers with the rest of the kernel) are also not solved by Cyclone. We therefore created our own dialect of the Cyclone programming language, known as ‘OKE-Cyclone’.

3) *The Bygwyn compiler*: The restrictions are enforced by a trusted compiler, known as *bygwyn* (named after a track by the Rolling Stones: ‘You can’t always get what you want, but you get what you need’). *Bygwyn* is customisable, so that in addition to its normal language rules, it is able to apply extra rules (customisations) as well. If restriction X is applied, a program is subjected to the compiler’s default rules *plus* the additional rules corresponding to restriction X. For example, we allow one to remove constructs from the language. If after such a restriction the compiler encounters the forbidden construct, it generates an error. In other words, depending on which extra restrictions we impose, the language that is permissible to the compiler may change.

The key idea is that the customisations to be used for a specific user’s program depend on the user’s role. In other words, users present their credentials to the compiler, and the credentials determine which customisations are applied. This is illustrated in Figure 3. Customisation types have unique identifiers, called customisation type identifiers (CTIDs). After compilation, *bygwyn* generates a signed compilation record containing both the CTID and the MD5 of the object code, explicitly binding the code to a type. Observe that the compilation rights are similar to

the loading rights, but that the two policies are decoupled, so that, theoretically, users may generate code that they will not be able to load. Given this, we allow security policies to be specified of the form 'a user with authorisation X is allowed to load code that is compiled with customisation Y'. Once loaded, the code runs natively at full speed, containing as many or as few runtime checks as necessary for this role.

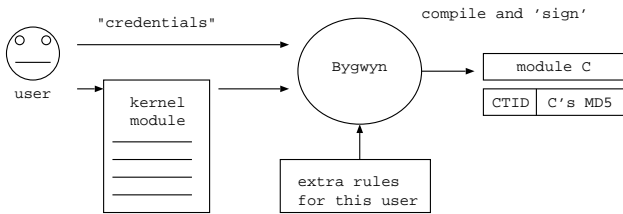


Fig. 3. User compiles kernel module

4) *Kernel level access*: Depending on the users' roles, they get access to other parts of the kernel directly, or via an interface to a set of routines which they may call (e.g., students in a course on kernel programming may get access to different functions than third-party network monitors). The routines are linked with the user code and reflect the role that is played by the kernel module. In other words, such routines are used to *encapsulate* the rest of the kernel, as illustrated in Figure 4. In the figure, some function calls (`foo`) are relegated to a wrapper, while others (`bar`) may be called directly. Access to data structures is regulated similarly.

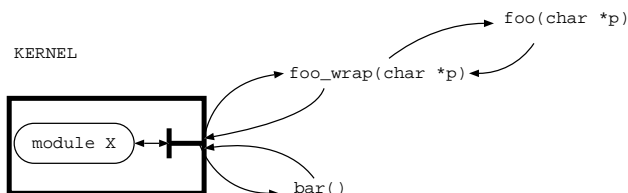


Fig. 4. The kernel is encapsulated behind an interface

We now briefly mention some of the mechanisms we implemented for making the OKE-Cyclone dialect safe for use in the kernel.

- 1) We perform (limited) global analysis of the code to decrease the number of dynamic checks.
- 2) Environment setup code (ESC) which contains the customisations is automatically prepended. It declares kernel APIs and other functions and variables and leaves the untrusted code with only the safe API (wrappers mostly). It also provides wrapper code for resource cleanup and safe exception catching. The ESC can configure this wrapping using a new `wrap extern` construct: *bygwyn* detects all potential entry points to the untrusted code and automatically wraps these functions using wrapper code

declared by the ESC.

- 3) Certain language constructs can be automatically removed from the language available to the programmer using a new `forbid` construct (examples include: `forbid extern "C"`, `forbid namespace`, and `forbid catch`).
- 4) A unique, randomly generated namespace is opened for the untrusted code to prevent namespace clashes and to prevent unauthorised imports of symbols from other namespaces.
- 5) The stack usage of untrusted code can be restricted to a usage limit defined in the ESC.
- 6) We are also able to limit CPU usage by using a modified timer interrupt. When a module has not finished withing its allocated time, an exception is thrown and the module removed. Code misbehaving in other ways is likewise removed.
- 7) We extended the region-based memory protection mechanism in Cyclone with a new kernel memory region `'kernel'`, to distinguish between kernel-owned and Cyclone-owned memory regions and implemented a simple mark-and-sweep garbage collector which ensures that pointers from the *OKE* modules to kernel memory are memory safe, and that freeing of module memory is handled correctly.
- 8) Specific fields of kernel structures shared with untrusted code can be statically protected by making making the structure members `locked`. Such members cannot be used in calculations, and cannot be cast to another type. Also, no other type can be cast to it, no pointer dereferences can take place, and no structure members can be read. Basically, locked types are limited to copying, and they cannot be read. This technique drastically reduces the need to anonymise data at run-time.

## B. Channels

The *OKE* channel elements all have simple interfaces that are implemented in either C or OKE-Cyclone (see Section III-A) and have been carefully designed in what is essentially an object-oriented fashion. For example, each channel element carries pointers to its own state, as well as to both blocking and nonblocking implementations of the pull and push operations. In this section we describe only a simplification of the main features of engines and queues. In essence, queues and engines communicate solely by pulling and pushing chunks of data from and to each other. A push or pull connection is typed, so that only specific items may be pushed or pulled on a connection between specific engines and queues. The types range

from simple types such as integers, and chars to composite types (e.g., IP packets).

1) *Queues, engines and their connections:* Queue behaviour in the default implementation is strictly FIFO (producer/consumer functionality on a circular buffer). More complex queueing schemes can be constructed using multiple FIFOs, or by providing an implementation of custom push and pull functions.

Queues are passive elements. They respond to `push` and `pull` operation, but they never initiate action themselves. In contrast, engines are active elements and hence have more methods in their interfaces. Apart from `push` and `pull`, engines provide a control/management interface (e.g. to `connect` and `disconnect` it to and from other elements), and a `run` method which is called when its is scheduled.

It was shown in Figure 1 that engines and queues can be connected and disconnected via simple control operations. Engines are connected to other engines or to queues by way of their `connect` methods. The control interface contains the `connect` methods needed to link engines to other engines or to queues. For this purpose they are provided with (among other things) the name of the element they should be connected to, as well as the port and the port direction (input or output). Queues do not provide such methods. Instead, they are managed by engines.

Engines and queues can be connected and disconnected at *runtime*. As such, the connections between them are not built into their logic. Instead, the control API allows explicit replumbing of the components. As it is dangerous to re-plumb an element when it is active (e.g., about to push a packet to the destination one wants to disconnect), these activities are protected with what is traditionally called a 'readers-writers' solution. In other words, many different data-path actions may be taking place at any time, but the management operations such as `connect` and `disconnect` require exclusive access.

2) *Crossing the domain barriers:* Engines and queues are tied to a *domain*. Currently, possible domains are: *userspace*, *kernel*, and *remote*. Elements in the same domain communicate by pushing or pulling simple types directly (call by value), or complex types by passing pointers (call by reference). As such communication within the same domain is efficient.

It is also possible to place engines and queues in different domains. For this purpose we use simple marshalling techniques similar to those used in remote procedure calls. For example, if engine  $E$  in domain  $D_1$  wants to push a network packet to queue  $Q$  in domain  $D_2$ , it really calls the `push` operation on a local proxy  $Q_{proxy}$  (also known as 'stub').  $Q_{proxy}$  is initialised with a set of routines that

enables it to connect to the remote implementation of  $Q$ . It marshalls the packet and initiates a 'remote' procedure call to push the packet on the 'remote' queue (note that 'remote' here means a different domain, which could easily reside on the same host). A push to a remote domain leads to the destruction of the data item on the local side.

Default proxies and marshalling routines have been written which are expected to suffice for almost all applications. The scheme can be easily extended, however, by writing new procedures for connecting to remote domains and for marshalling the data.

3) *OKE Corral packet traversal:* Once a packet is classified (by the classifier in Figure 1) as belonging to the AN, it will be pushed on the AN's entry engine and follow the data-path determined by the AN's engines and queues. If necessary, some of the fields in the packet may be protected against read and write access violations using the `locked` keyword described earlier (note that locked fields cannot be pushed across domains). The entry engine pushes the packet to the next engine and so on, until one of the following three events has occurred: (1) the packet is dropped, (2) the exit engine is reached and the packet has been sent, or (3) an intermediate queue has been reached.

### C. The Active network

The AN runtime is derived from a home-grown active network, which is capable of running either a Java or a Tcl execution environment. For the *OKE Corral* implementation we have limited ourselves to the Tcl implementation. The goal of the runtime is to provide a very simple environment for AN experiments. Loading code onto the runtime can be done either out-of-band (using an explicit load operation), or in-band, in the form of capsules.

The runtime consists of an interpreter and a fairly extensive set of operations that are specific to the AN. This is called the *core set* of operations, all of which are implemented in C. The core set contains elementary operations, e.g. a repertoire of functions for convenient access to received packets and for finding the load on specific links, etc. It also contains a `send` operation for pushing a packet out onto the network. Packets are stored in packet buffers, of which there is a fixed number. One of the buffers is designated the 'current' buffer and this is used to receive the next packet in. A small number of operations in the core set is responsible for managing the buffers, e.g. to set the current buffer, to execute `safe memcpy` and `memmove` operations, etc. Finally, there is an additional library that is fully implemented in Tcl. This package contains a large number of functions that are commonly used, as well as wrappers around the functions around the core set.

1) *AN and channels*: The runtime back-end was modified to sit on top of the *OKE* Channels. More correctly, by implementing the engine interface, the runtime really becomes an engine  $E_R$  itself.  $E_R$  initialisation code disconnects the packet entry and exit engines assigned to it and reconnects the entry engine to a kernel-domain queue. It also connects  $E_R$  to the other end of the queue for inbound traffic and to the packet exit engine for outbound traffic.

After initialisation, it is the active code in the runtime that is responsible for the management and control of the engines and queues in its channels. For example, operations were added to the AN's repertoire to allow it to connect or disconnect all elements under its control. Depending on the AN, bootstrap kernel modules containing pre-installed engines and queues may be loaded at initialisation. The components in the modules can be freely used by the AN to construct new data-paths. They may be highly efficient, e.g. written in C and not containing any checks whatsoever (after all: since we provided them, we know they are safe).

There are also commands to enable the active code to add entirely new components (engines and queues) to the data-path. In the following discussion we will assume that the target domain for the new components is the kernel, since this presents the most severe security risks. For the purpose of loading data-path components, the active code is allowed to pull a module containing them from a remote webserver. Similarly, it is allowed to refer to webpage for the credentials. The module together with the credentials is then offered to the *OKE* codeloader. Provided the credentials are valid, the codeloader pushes the module into the kernel. At that point the codeloader is able to manage the new engines and queues in exactly the same way as the pre-installed components.

2) *Discussion*: When loading new components in the data-path, as discussed in the previous section, safety was guaranteed by the *OKE*. This means not only that the code *must* be written in *OKE-Cyclone*, but also that the compiler that compiled this code prior to loading must be *trusted*. We have not addressed the issue of whether and under what circumstances compilers in remote domains can be trusted. We call this the 'trust propagation' problem. One possibility is to have a well-known group of trusted compilers that can be used to generate object code with compilation records that are accepted by many sites (the "VeriSign model"). Alternatively, we could store the code in source format and have a *local* (and presumably trusted) compiler generate the object code anew just prior to loading it. We are currently exploring and evaluating these and other solutions.

Another issue concerns the authorisation of requests to load code in the kernel. Normally, when a client tries to load an *OKE* module in the kernel it is required to authenticate every such request by signing a number of items with its private key. However, if the code is running on an unknown active node, the client may be reluctant to send its private key there for fear of it being copied. In the model that was adopted in the *OKE Corral*, we have 'single sign-on' behaviour. In other words, the identity (public key) of the client is established at the time the active code is loaded on the runtime. From that point onwards, this is the identity that is used in `load`, `connect` and other requests. The active code is not required to sign anything. It should just find and present the right credentials.

#### IV. RESULTS

We don't think that the number of packets per second that can be handled is a relevant measure in evaluating the *OKE Corral*, for two reasons. First, at high speeds, such numbers generally say more about the implementation of the traffic capture than about packet processing. For example, on a software router as implemented in the Click-router the number of packets per second varies greatly depending on whether the packet capture is interrupt-driven or polling [SOK01]<sup>1</sup>. Second, we are really more interested in how the *Corral* compares to typical AN implementations and this concerns primarily the nature of the code execution: in-kernel native code, versus interpreted code (often executing in userspace). Readers interested in the performance in terms of packets per second that potentially can be achieved with a channel-based system should refer to [CM01].

Instead, we evaluate the *OKE Corral* by measuring the performance of the data-path components and by comparing the results with alternative implementations. All measurements described in this section were taken on a PIII 1GHz PC running a Linux-2.4.18 kernel. The overhead of a push from entry engine to exit engine without any processing takes approximately 250 nsecs (this includes all locking and sanity checks). The applications used for the comparison are in the domains of transcoding (application *T*) and monitoring (application *M*). Both applications are considered components on the data-path. In the *OKE Corral* version of the experiment, they are implemented in *OKE-Cyclone*, and dynamically loaded in the kernel by the active control code in userspace. *M* implements a packet sampler which is meant to push 1% of all

<sup>1</sup>The NAPI patch for Linux turns an interrupt-driven kernel into a polling one, whenever the load gets high.

packets on a queue which can be read by a monitoring application in userspace (in this case, this is the AN) using a pull operation. On (pull) request, it will also report the total number of bytes of all packets that passed through  $M$  since the last report.  $T$  resamples audio packets to a lower quality (containing half the bits) and thus works on the entire payload. For this reason,  $T$  also requires a recalculation of the checksums in both the IP and UDP header. In previous work, we discussed a forward error correction transcoder and showed that its performance in the *OKE* is only marginally worse (10% overhead in the worst case) than that of a pure C implementation [BS02].

In the current experiment, both types of applications may operate on the same packets. In fact, there are 4 types of packet, all of which are UDP with destination ports  $p_0$ ,  $p_1$ ,  $p_2$ , and  $p_3$ . The experiment is illustrated in the leftmost illustration of Figure 5. Packets for port  $p_0$  are subject to both transcoding and monitoring. Packets for  $p_1$  are subject to transcoding but not to monitoring, i.e. they are pushed directly to the exit engine by the transcoder engine. Destination  $p_2$  packets will pass *through* the transcoder, but are not touched by it. Instead they are moved straight to the monitoring engine. Packets for  $p_3$  are neither resampled nor monitored, but do pass through the entry engine, the transcoder and the exit engine.

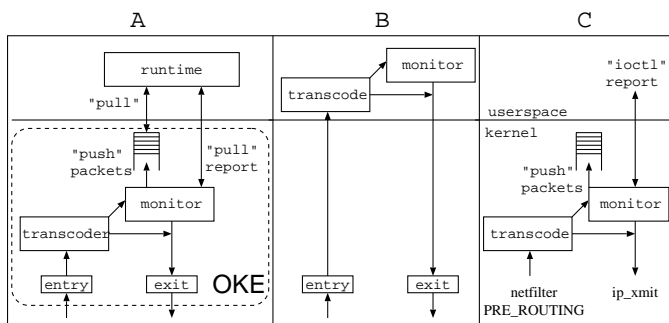


Fig. 5. The three implementations of the applications

We evaluate 3 different implementations: (A) all components in *OKE*, (B) all components in AN runtime, and (C) all components in in-kernel C, as shown in Figure 5. All three versions are possible in the *OKE Corral*, but we are most interested in solution (A), as it provides maximum flexibility while still running natively in the kernel. We measure time between packet entry at the Linux netfilter hook to the time that we send the packet (or queue it for userspace).

The results are shown in Figures 6-8. As expected, we see in all figures that the overhead for  $p_0$  and  $p_1$  type packets is strongly dependent on the packet size. The  $p_2$  and  $p_3$  on the other hand are basically flat, as there is not

even need to recalculate checksums if the packet doesn't change. Also notice that in the Tcl implementation the effect of monitoring is no longer visible due to the enormous overhead introduced by the Tcl interpreter and our (admittedly very inefficient) context switching.

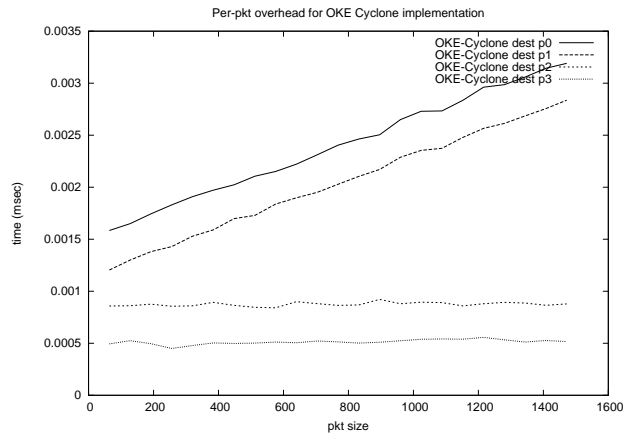


Fig. 6. Scenario A: all in the kernel in OKE-Cyclone

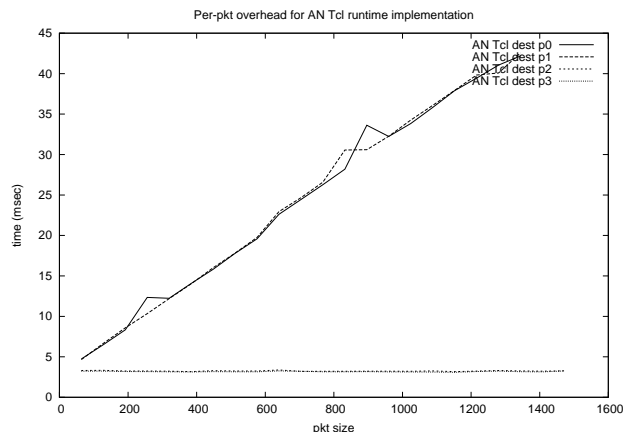


Fig. 7. Scenario B: all in userspace AN runtime

For none of these solutions have we attempted any manual optimisation. Moreover, it is clear that there exist much faster AN runtimes than the Tcl environment that we have used. However, in previous work we measured that a copy from kernel to userspace using a direct `ioctl` pipe to the kernel takes approximately 0.002 msecs in the best possible case, and considerably longer if we use `libipq` (we measured 0.008 msecs on average for a copy from kernel to userspace). If a copy to userspace is needed, it will be difficult (if not impossible) to optimise away this overhead. A copy back to the kernel takes approximately the same amount of time, so regardless of the speed of the C code, we lose 0.004 msecs, just on the copies. As shown in Figure 6, this overhead alone exceeds the total time needed by the *OKE-Cyclone* implementation. Even so, we are currently investigating faster userspace implementations.

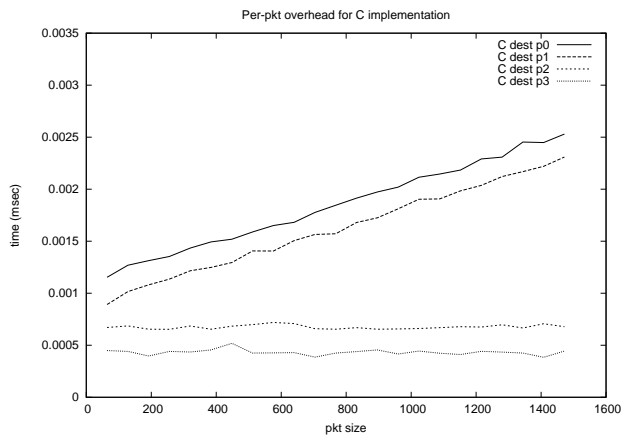
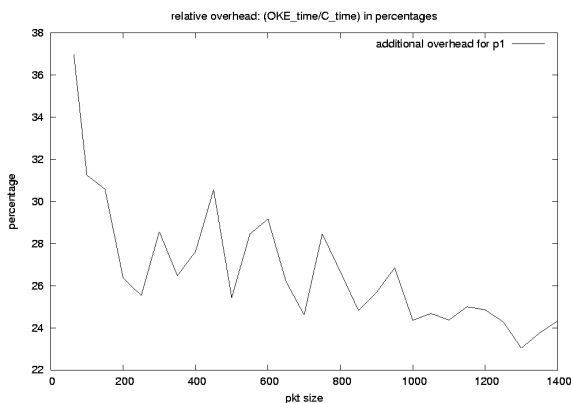


Fig. 8. Scenario C: all in the kernel in C


 Fig. 9. Overhead of processing packet  $p_1$  in the OKE compared with C

In Figure 9 we also plot the relative overhead of performing the transcoding application in the *OKE* instead of native C. Concretely, the figure plots the ratio computed by  $(\frac{T_{Cyclone}}{T_C} * 100 - 100)$  for the  $p_1$  packet times shown in Figures 6 and 8. It is interesting to note that the overhead per byte decreases as the packet size increases. This is caused by the fact that the fairly substantial one-time overhead is amortised over a large number of bytes. The overhead of the implementation with the AN in the datapath is orders of magnitude and therefore not plotted.

For now, we conclude that the difference in performance between the AN implementation and either of the other two implementations is orders of magnitude. Between the *OKE* and the ‘pure C’ implementation the difference is approximately 25%. The results suggest that a substantial gain in performance can be achieved by employing something like the *OKE* in ANs. If the speed of pure C is required, active code is still able to control and manage these components, and to build new applications by clicking together elements from a predefined set.

## V. RELATED WORK

Organising AN software in a hierarchical fashion is advocated in many active network projects, e.g. SwitchWare [AHK<sup>+</sup>98]. Such approaches differ from the *OKE Corral* in that they are mostly concerned with (interpreted) user space code for all loadable extensions. Clicking components together to form channels is equally common in ANs. A good example is CANEs, which allows extensions to be injected in predefined locations on the data-path [MBC<sup>+</sup>99]. A third aspect, the separation of control and data path in programmable networks has also been advocated in a number other projects, e.g. SwitchWare at UPenn and the work on programmable network control in Cambridge [BIML01].

Many projects target safety in operating systems (OSs). These include language-based approaches such as BSD Packet Filters [MJ93], proof carrying code [NL96] and software fault isolation [RSTS93], as well as OS-based approaches such as Nemesis [LMB<sup>+</sup>96], ExoKernels [EKO94], and SPIN [BSP<sup>+</sup>95]. Trust management combined with module thinning in ANs was introduced in the Secure Active Network Environment [AHK<sup>+</sup>98]. The combination of trust management and AN code loading was also discussed in [HK99]. An exhaustive discussion of these projects is beyond the scope of this paper. In short, the *OKE* provides a more complete safety model than SFI which is simpler than PCC and distinguishes itself from such approaches as Nemesis, Exokernels and SPIN in that it is implemented on a commonly used OS. Interested readers are referred to the discussion in [BS02].

In the remainder of this section, we will compare our work briefly with a number of other systems that support the loading of native code in the kernel of an operating system, by looking at how well they support the following ten features targeted by the *OKE Corral* (and as described in this paper):

- 1) The system explicitly supports 3rd party code in the kernel.
- 2) The kernel is fully programmable, although if needed, we are able to restrict access to specific APIs, data, etc., at compile time.
- 3) Resource control is enforced for CPU, memory, etc.
- 4) Safety is enforced in the sense that a module is not able to crash, dereference NULL pointers, inadvertently free kernel memory it points to, etc.
- 5) Data channels are composed of LEGO-like components (like in Click).
- 6) Configuration of these channels is possible at runtime.
- 7) Data and control are explicitly separated.
- 8) AAs in the form of capsules are able to configure the

	SILK	ANN	PromethOS	SPIN	FLAME	Click	OKE Corral
1	++	--	-	++	++	--	++
2	+/-	+/-	+	++	-	+/-	++
3	+	-	-	+	+	-	++
4	--	-	-	++	+	--	++
5	+	-	-	-	-	++	++
6	++	+	+	++	+	-	++
7	++	-	++	0	+	+	++
8	-	-	-	0	0	0	++
9	++	+	++	0	++	+	++
10	++	++	++	--	++	++	++

TABLE I

*OKE Corral* FEATURES MENTIONED IN THE TEXT COMPARED WITH OTHER SYSTEMS. SYMBOLS: '+' = STRONG SUPPORT, '-' = WEAKER, '+/-' = PARTLY, '0' = NOT APPLICABLE TO THIS SYSTEM.

data channels to the point of loading and connecting new native code components.

- 9) Out-of-band loading of AAs in the kernel is supported.
- 10) The system is implemented on a common OS.

Note that we do not aim for a true comparison of these very different systems. We only look at how well other approaches support some of the more attractive features of the *OKE Corral*. The results of the comparison (using the same numbering of features as above) are shown in Table I. Below we discuss the projects mentioned in the table.

We have been strongly influenced by the Click-router project which uses a simple LEGO-like organisation of forwarding code to build a high-performance router in software [CM01]. Although we didn't use the Click code directly, we implemented a very similar system (in C). However, whereas Click components are assumed to reside in the same domain (e.g. the kernel), we permit them to be distributed at will over kernel, user-space and even remote machines.

Our processing hierarchy resembles that of the 'extensible router' [NLA<sup>+</sup>02]. In particular, SILK also provides fast kernel data-paths with support for resource accounting. However, it does not provide safety. The code loading in our work somewhat resembles that of ANN [DPP99]. In ANN active code is replaced by references to modules stored on code servers. On a reference to an unknown code segment in a node, the native code is downloaded, linked and executed. Similarly, a recent project called PromethOS described elsewhere in these proceedings, supports kernel plugins with explicit signalling for plugin installation [RLAB02]. Neither approach targets safety as aimed for by the *OKE*. In a sense, the *OKE* is providing a safe environment for allowing third-party kernel plug-ins in addition to the pre-defined ones.

SPIN, which builds on the safety properties of Modula-

3, is close in spirit to the work presented here. However, unlike the *OKE*, SPIN does not control the heap used by 'safe' kernel additions. Additionally, it is not a commonly used OS.

Early work on the use of the Cyclone for kernel work and KeyNote for policy control was demonstrated in FLAME [AIM<sup>+</sup>02] which is similar to the *OKE* and a good example of how similar principles are used for different goals. FLAME is aimed at safe network monitoring and not on fully programmable kernels. In contrast, the *OKE* provides the necessary features for general-purpose kernel extensions, with a focus on customisability. FLAME provides little flexibility in the restrictions placed on a module, and full interaction between the module and the kernel (e.g., using pointers) is not allowed. While essential to the *OKE*, neither of them are needed in FLAME.

## VI. CONCLUSIONS

In this paper, we have presented the *OKE Corral*, an AN environment that builds on a set of technologies recently developed in the research community to achieve high-speed programmable packet processing in an active node. A model similar to that of the 'Click' router was used to construct highly efficient data-paths of which the components can be loaded and controlled by active code at runtime. In contrast with most AN implementations, the code can be loaded anywhere in the processing hierarchy, from the AN runtime to the kernel. The open kernel environment ensures safety in such a way that even fully compiled and optimised code can be loaded into the kernel. In the *OKE Corral* the 'active code' running in the AN runtime plays the role of control and management software and operates at a much slower speed than the fully compiled code in the data-path. Both the control and the data plane use the same *OKE* channel mechanism to construct their flows.

The performance of the *OKE Corral* varies with the amount of programmability. At one extreme, only the control and management is programmable, while the data-path consists of predefined and highly optimised 'standard' components based on which custom data-paths can be constructed. At the other extreme, there is the 'capsule' approach advocated in some other AN projects. Between these two extremes, but closer to the former extreme, we have the *OKE* channels approach. For maximum flexibility, the different kinds of programmability may be mixed and matched, so that 'capsules', pre-defined and third-party components all interact to build data and control flows. Open issues include the problem of heterogene-

ity, as well that of trust propagation. These are the topics of ongoing research.

#### ACKNOWLEDGEMENTS

We are indebted to Lennert Buytenhek for helping to sort out some of the thornier issues in kernel hacking.

#### REFERENCES

- [AHK<sup>+</sup>98] D. Scott Alexander, Michael Hicks, Pkaj Kakkar, Angelos Keromytis, Marianne Shaw, Jonathan Moore, Carl Gunter, Trevor Jim, Scott M. Nettles, and Jonathan Smith. The SwitchWare active network implementation. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.
- [AIM<sup>+</sup>02] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, Michael B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proc. of NOMS'02*, April 2002.
- [BFIK99] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system version 2. *Network Working Group, RFC 2704*, September 1999.
- [BIML01] Herbert Bos, Rebecca Isaacs, Richard Mortier, and Ian Leslie. Elastic networks: An alternative to active networks. *JCN (Special Issue Programmable Switches and Routers)*, 3(2):153–164, June 2001.
- [BS02] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *Proceedings of OPENARCH'02*, New York, USA, June 2002.
- [BSP<sup>+</sup>95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc of SOSP-15*, pages 267–284, 1995.
- [CM01] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proc. of USENIX Annual Technical Conference (USENIX '01)*, pages 333–346, Boston, Massachusetts, June 2001.
- [DPP99] D. Decasper, G. Parulkar, and B. Plattner. A scalable, high performance active network node. *IEEE Network*, January 1999.
- [EKO94] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The exokernel approach to extensibility. In *Proc. of OSDI'94*, page 198, Monterey, California, November 1994.
- [HK99] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. Extended version at <http://www.cis.upenn.edu/~switchware/papers/secureplan.ps>.
- [JMG<sup>+</sup>02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX 2002 Annual Technical Conference*, June 2002.
- [LMB<sup>+</sup>96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *JSAC*, 14(7), September 1996.
- [MBC<sup>+</sup>99] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and canes: Implementation of an active network, 1999.
- [MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, San Diego, Ca., January 1993.
- [NL96] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, Seattle, Washington, October 1996.
- [NLA<sup>+</sup>02] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb, S. Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible routers for active networks. In *DARPA AN Conference and Exposition*, June 2002.
- [RLAB02] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A dynamically extensible router architecture for active networks. In *Proc. of IWAN 2002*, Zurich, Switzerland, December 2002. Springer.
- [RSTS93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault-isolation. In *Proc. of SOSP'93*, pages 203–216, December 1993.
- [SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *USENIX*, November 2001.