# Dynamically extending the Corral with native code for high-speed packet processing

Herbert Bos [a,*] Bart Samwel [b] Ilja Booij [b]

[a] *Computer Science Department, Vrije Universiteit Amsterdam*
*De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands*

[b] *Leiden Institute of Advanced Computer Science*
*Niels Bohrweg 1, 2333 CA Leiden, The Netherlands*

**Abstract**

By combining the Open Kernel Environment, a Click-like software model known as Corral and basic concepts of active networking, we allow third-party code to control the code organisation of a network node at any level, including kernel and network card. We show how an active network environment was implemented and how this environment allows slow active code to control the code organisation of the fast path. The underlying code is structured much like components in a 'Click'-router that may be connected or disconnected at runtime. Not only are active packets permitted to reconfigure predefined native components in the networking code, by using the safe programming model of the open kernel environment they are also able to load and link their own native components at any place in the datapath and at any level in the processing hierarchy.

*Key words:* open kernel environment, active networks

## 1 Introduction

For reasons of safety, programmability of a node by third party code (e.g, for purposes of network monitoring or active networks) tends to be implemented by sandboxing such code in user space. Moreover, the code is often interpreted in a virtual machine. Even in non-active environments interpreters are frequently

* Corresponding author
*Email addresses:* HerbertB@cs.vu.nl (Herbert Bos), bsamwel@liacs.nl (Bart Samwel), ibooij@liacs.nl (Ilja Booij).

used, especially when 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 special-purpose language that is known to be safe and that is interpreted in the kernel [MJ93].

Interpreted solutions, however, tend to be slow. Moreover, in this paper we argue that better alternatives exist. In previous work, we have already 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 [BS02a]. 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 significantly outperforms implementations that filter packets in the kernel while processing them in userspace.

In this paper, we describe the implementation of the *OKE Corral* (Code Organisation and Reconfiguration at Runtime using Active Linking), an environment that allows one to extend and modify a node's software configuration in all processing levels (e.g. kernel, userspace) at runtime. As a practical example, it is shown how the *OKE Corral* was used to build a high-speed active network (AN). We emphasize that ANs serve only as an example target domain. The framework can be used in any situation where safe and convenient programming of the kernel for purposes of fast packet processing is required.

The contribution of this work is that three novel technologies in the field of networking and open systems (the OKE, a Click-like software model and active networks) are combined to provide a platform for fast programmable packet processing with explicit separation between control and data flow. Since our first presentation of the *OKE Corral* in [BS02b], we have extended the system to make it both more open and more extensible. This paper describes the current state. In the *OKE Corral* high-speed packet processing is managed by slow-speed control code. The key features are summed up as follows:

(1) We borrow the LEGO-like software organisation that is advocated in the 'Click' router project [CM01] both to build fast data-paths and to implement paths for control traffic. This component is known as the *'Corral'* and, as will be shown later, differs from Click in various aspects.
(2) Currently, one of the components on the control path is an AN runtime in user space (if required, the control path *can* also be used for data packets, but because the control path is fairly slow, this is much less efficient).
(3) Configuration and implementation of all remaining control and data path components can be initiated both by active packets or remote parties.
(4) The *OKE* is used to ensure that kernel-level implementations of the path

components are safe.

The idea is that slow-speed third parties can configure the software configuration of a host's fast path by replumbing a Corral configuration (e.g., by employing active packets). Moreover, the third parties may cause new elements to be introduced in a running configuration. Such elements consist either of unrestricted C code or of resource controlled code running in the *OKE*. Which type of code should be used when depends on the programmer's privileges and on the potential harm that can be caused by the element.

Although many of the individual components are not 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) full programmability of both kernel and user space with optimised native code, (b) while still providing resource control and safety with respect to memory, CPU, available API, etc., (c) in addition to 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 more recent RBClick project of the University of Utah has similar goals as the *OKE Corral*, but differs in important aspects, as explained in Section 5.

The *OKE Corral* described in this paper has been fully implemented and a prototype implementation has been evaluated (see Section 4). Both the *OKE* and the Corral can be downloaded from `http://www.liacs.nl/~herbertb/projects/{oke,corral}`. 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 2, and the prototype implementation of the architecture in Section 3. The protoype is evaluated in Section 4. Related work is discussed in Section 5, and conclusions are drawn in Section 6.

## 2 Architecture

As indicated by the numbers in Figure 1, the *OKE Corral* implementation as discussed in this paper builds on three technologies: (1) the open kernel environment, (2) one or more AN runtimes, and (3) the Corral, i.e., packet channels that implement control and data paths and link the various components seen by the packets as they traverse the network node (e.g., processing elements and queues, represented by the black boxes in Figure 1). The details of all elements of Figure 1, as well as the way in which they interact will be explained below. We should mention that the configuration in Figure 1 is an example configuration only. As the OKE Corral provides an environment to build efficient packet processing environments (such as active nodes), rather than a full-fledged active node in its own right, the configuration of engines and queues may vary from application to application (and indeed from active network to active network). And so while we stress that the *OKE Corral* can be applied in many different domains, for clarity we will assume in the remainder of this paper that the application domain is ANs. To the right of our example AN architecture in Figure 1, we have indicated an 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).

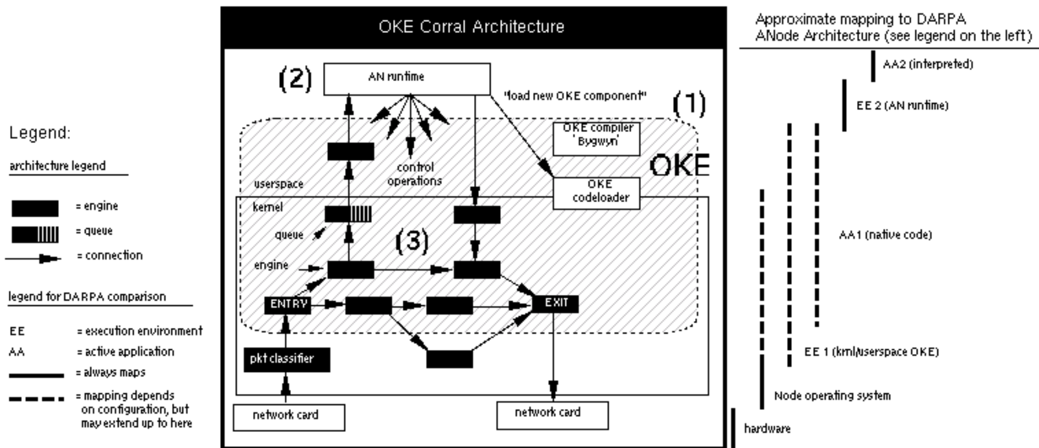



Fig. 1. Overview of the *OKE Corral* (example configuration only).

### *2.1 Corral terminology*

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

Processing elements in the *OKE Corral* are known as 'engines' and 'queues'. Queues, such as FIFOs and stacks, are static: they only function as queues and wait for data to be pushed to and pulled from them. Engines, on the other hand, are active elements and implement a `run()` method. An example is the `pump` element described below. Engines and queues have multiple input and output ports that may be logically attached to other elements to form *connections* (drawn as arrows in Figure 1). By adding or deleting elements and connections in a running system, a Corral application can be changed at runtime. Both the framework and the elements are implemented in ANSI C in an 'object-oriented' fashion. A connection between two elements is possible only when the output from the source port and the input from the destination port are of the same *type*. The type can be a simple type (e.g., an integer, or a float), or a complex type (e.g., a `struct`, or a packet). Different ports of an element can be used for different types. For instance, an element that receives IP packets on one of its input ports and sends them out on output port 1, may use output port 2 to report the (integer) packet count. Connections are point-to-point only.

*Data transfer*

Elements exchange data over the connections. 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. Elements are normally also either 'push' or 'pull', but hybrid forms, known as *pumps* and *queues* are also possible. A pump is a 'pull-to-push' element, i.e., it pulls data from a source element and pushes it to a destination element. Thus a pull is changed into a push.

The inverse of a pump is a queue. A queue accepts pushes (of data items) on its input, and pulls on its output. Most queues may be filled and emptied by more than one element. As shown in Figure 1, elements are connected and disconnected at runtime 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 element, and the exit element is a channel. Channels consist of "subchannels", which start at an engine and end either at a queue, or at the packet exit element. 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 order to reduce complexity in configurations with many elements, we are currently working on aggregation at the level of subchannels, i.e., treating entire subchannels as a single element with a unique identifier. However, this

is not yet fully implemented.

*Element classes*

All elements in the Corral are instances of element 'classes' that exist in the Corral. If an element class does not exist at a specific host, an element of that class cannot be instantiated. It will be shown later that element classes can be loaded and unloaded at runtime. Elements can hold data and memory to hold this data is generally allocated when the element is instantiated. Different instances of element classes are independent; they use the same functions, but do not share data (although data sharing can be easily programmed into an element class).

*Code domains*

In contrast to the Click approach, elements may reside in the kernel, on a network processor, in user space, or even on remote machines. Wherever they reside is known as the element's "domain". Similarly, they may exist either inside the *OKE* (in which case they are subject to checks and resource limits, as explained later), 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 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.

### *2.2 OKE, Corral and AN runtime 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 pushes them 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.

*Standard element classes and new ones*

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) safety 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. There are two possibilities for loading code in the kernel.

**'Normal' elements** First, given the appropriate privileges (in the form of credentials), users may load 'normal' element classes, written in C and compiled by any C compiler. For safety, rhe Corral provides fine-grained admission control. For example, credentials may specify that only certain classes can be loaded by a user (elements and element classes are uniquely identified by their MD5 or SHA-1 checksums), and that instantiations of a class may only be connected to specific other elements that are also owned by this user, and so on. Credentials are signed by an 'authority' and may be delegated. In other words, a host's local policy $P$ may permit a user $U_1$ to perform all operations in set $S_1$. $U_1$ may now create a credential $C_1$ for user $U_2$ to perform all operations in $S_2 \subseteq S_1$. $U_2$ may in turn create a credential $C_2$ for $U_3$ to perform operations in $S_3 \subseteq S_2$. As long as $U_3$ presents both $C_1$ and $C_2$, he/she will be able to perform all operations in $S_3$. Using the identity of $S_3$ and the unforgeable credentials generated by $S_1$ and $S_2$, the admission control system is able to check that the request complies with $P$.

In general, the ability of a party to load such 'non-sandboxed' elements in the kernel should depend on the extent to which this party is trusted and on the potential harmfulness of the code. For instance, it is probably acceptable to provide most users with credentials for loading a well-known and safe packet counter, to be connected to other elements owned by this user. On the other hand, the permission to load completely unknown elements should only be given to highly privileged users, such as the system administrator.

**OKE elements** Second, in the *OKE Corral*, even unknown native code from an untrusted party can be loaded in the kernel as long as the *OKE* is used. The *OKE* is able to restrict code according to the code-loading party's privileges. Privileges are again 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 3.1.

### *2.3 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 channels are also used to implement slow-speed control paths which commonly lead to AN runtimes in user space (or even on remote hosts) and which are used to carry the active packets. The active packets contain 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 programmable networks are to scale to the size of the Internet.

## 3 Implementation

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

### *3.1 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.

Recently, the *OKE* mechanisms were tested on network processors [BS03]. The complete system allows third parties to load code not only in the kernel, but also on the microengines of an IXP1200 network processor.

In this paper we only give a high-level overview of the kernel version of *OKE*. A detailed explanation is presented in in [BS02a] and the work on network processors is described in [BS03]. 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 *bygwyn* compiler, which compiles user code according to the rules corresponding to the user's privileges. Both are also shown in Figure 1.

#### *3.1.1 Codeloader*

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 CL is used for loading both 'normal' and '*OKE*' Corral element classes. The process is illustrated in Figure 2.

The trust scheme and authorisation checks are implemented using KeyNote [BFIK99] and the OpenSSL library. 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 as described in Figure 2. Privileges 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 loading rights are only valid during office hours. A module type is instantiated when source code corresponding to the type is compiled. The trusted *OKE* compiler generates an unforgeable 'compilation record' which proves that module $M$ (identified by its MD5 or SHA-1) was compiled as type $T$ by this compiler.

#### *3.1.2 Language issues*

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.

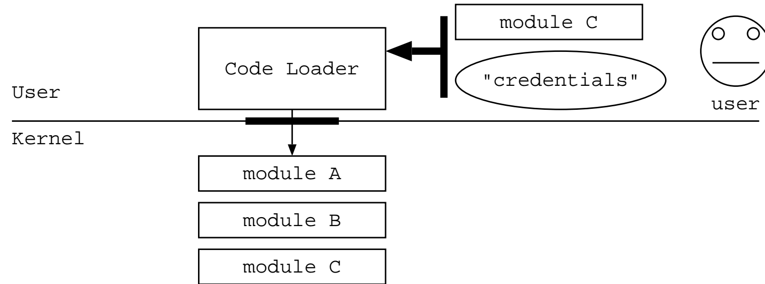


Fig. 2. User loads module in the kernel

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 ‘little 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 language is preferable 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 type-safe version of C which ensures safe use of pointers and arrays, offers fast, region-based memory protection, and inserts few runtime checks [JMG+02]. However, for true safety and speed, we needed both more and less than what is 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’.

#### *3.1.3 The Bygwyn compiler*

The restrictions are enforced by a trusted compiler, known as *bygwyn*[1]. *Bygwyn* is customisable, so that in addition to its normal language rules, it is able to apply restrictions on resource usage. If restriction X is applied, a program is subjected to the compiler’s default rules *plus* the additional rules corresponding

[1] Incidentally, the name derives from: ‘You can’t always get what you want, **B**ut **Y**ou **G**et **W**hat **Y**ou **N**eed’, by the Rolling Stones, which seems appropriate.

to restriction X. The restrictions may include language constructs. For example, some language constructs may be removed from the language entirely. If after such a restriction the compiler encounters the forbidden construct, it generates an error. Moreover, the *OKE* controls the resource usage of the module at runtime (e.g., in terms of CPU, stack and memory usage, APIs, etc.)

The key idea is that the resource restrictions to be used for a 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/SHA-1 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.

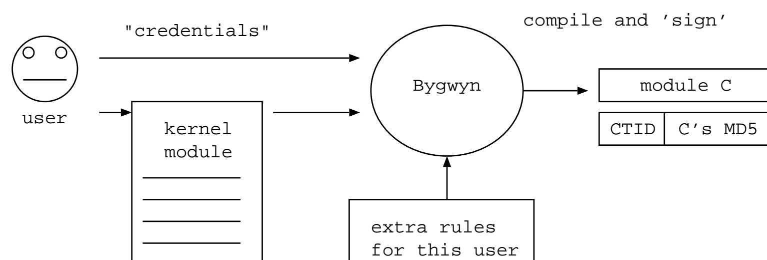



Fig. 3. User compiles kernel module

### *3.1.4 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 compiled and 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.

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

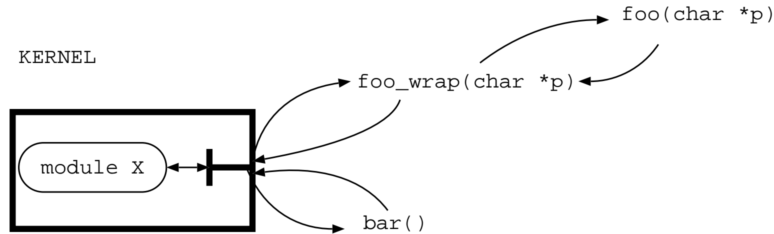


Fig. 4. The kernel is encapsulated behind an interface

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. All functions in the *OKE* module that may be called can be wrapped *automatically*. The ESC configures 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 X`, and `forbid catch Y`).
(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) The *OKE* is also able to limit CPU usage by using a modified timer interrupt. When a module has not finished within 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 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.

### 3.2 Corral

The concept of clicking kernel components together to create new functionality is a tried and useful practice. The *x-Kernel*, first proposed in the late 80s, provided mechanisms to statically stack network protocols in this way [HP91]. Similarly, the *STREAMS* abstraction, proposed even earlier, allowed protocol stacks to be composed dynamically [Rit84]. This work was influenced by all such approaches and in particular, as mentioned earlier, by the Click software router. The Corral elements all have simple interfaces that are implemented in either C or OKE-Cyclone (see Section 3.1) 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 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).

#### 3.2.1 Queues, engines and their connections

In the default implementation there are only two types of queues: FIFOs and stacks. The default queue is a simple FIFO with 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()` operations, but they never initiate action themselves. In contrast, engines are active elements and hence have more methods in their interfaces. In addition to `push()`, and `pull()`, all engines provide a control/management interface (e.g., with `connect()` and `disconnect()` methods to link to and from other elements), and a `run()` method which is called when it is scheduled. An element's `connect()` method is very simple: it is provided with (among other things) the name of the element that it should be connected to, as well as the port and the port direction (input or output). If the ports are of the same type, a connection between the elements is established.

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. While it may seem dangerous to replumb an element when it is active (e.g., about to push a packet

to the destination one wants to disconnect), these activities are protected so that connections are always able to 'drain'. While this is happening the two elements may already have been disconnected, i.e., only the pipe itself needs to be kept while it is emptied - the source element itself may even already have been destroyed.

### *3.2.2 Crossing the domain barriers*

Engines and queues are tied to a *domain*. Currently, possible domains are: *userspace*, *kernel*, and *remote*. A preliminary implementation of Corral domain *'IXP1200 network processor'* has also been written and will be discussed in Section 3.4. 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 takes place within the same address space, interaction within a domain is quite 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.

### *3.2.3 Corral extensibility*

Even ignoring the *OKE*, the Corral is extensible in the sense that new element classes can be loaded and instantations of these classes can be created at runtime. Element classes to be loaded are specified by a URL which points to a file in an element class repository. The repository can be hosted on any machine reachable via HTTP or FTP. Providing a repository is as simple as hosting a webserver of FTP server, as the repository itself performs no access control or security. All admission checks are devolved and all users with write access to a repository can make their element classes available to other users. The corral uses the `libcurl` library for downloading the element classes. The following control actions are subject to (local) admission control:

- loading and unloading of element classes;
- instantiating an element of a class;
- deleting an element;
- connecting two elements;
- disconnecting two elements

Without the appropriate credentials, none of these operations are permitted. Loading an element class also fails if the class is already loaded, or cannot be found at the specified repository. Unloading an element class fails if the class is not present at the Corral site, or if there are still instances of this class running in the Corral. Obviously, the implementations of class loading in domains 'kernel' and 'user space' are somewhat different.

For example, for userspace the dynamic library operations `dlopen()`, `dlsym()`, etc., are used. In essence, this means that a new element class is implemented as a dynamically loadable library. For the kernel, new Corral classes are implemented as kernel modules and we rely on the dynamic linking of such modules that is already present. Again, the codeloading facilities of Linux are wrapped so that ordinary users are only able to load modules that have been approved by the admission check in the code loader.

For each of the control operations above, certain *conditions* may be specified (e.g., by the system administrator) and these will be checked when the user attempts to perform the operation. These conditions are specified in the credentials presented by the user. For instance, a condition may specify that a specific user may only load an element in one specific domain (say, 'userspace'). Examples of properties that may be checked in conditions are:

- **user identity:** a user is identified by his/her public key (a nonce challenge is used to ensure this request came actually came from this user and is also not a replay);
- **host:** maybe an operation is only permitted on certain hosts;
- **domain:** some actions may be allowed only in a specific Corral domain;
- **class identity:** certain rights may pertain only to specific element classes;
- **number of loaded classes:** a limit may be set on the number of classes a user is allowed to load;
- **number of class instances:** a limit may be set on the number of instances of classes a user is allowed to create;
- **ownership:** some operations may only be permitted on the user's own instances;
- **class loading party:** one may for instance specify that a class can only be unloaded by the party that loaded it;

Furthermore, there is a whole set of properties that can be used in conditions that pertain to creating and deleting connections. These include, for instance,

```
KeyNote-Version: 2
Comment: Within the application 'Corral', the
         authorizer grants the licensee the right
         to load and unload a pkt_counter class.
Local-Constants: LKEY = "rsa-base64:FAKE_KEY_USR"
Authorizer: "rsa-base64:FAKE_PUB_KEY_AUTHORIZER"
Licensees:  LKEY
Conditions: app_domain == "Corral" &&
            corral_Host == "corralhost.liacs.nl" &&
            (corral_action == "load" ||
             (corral_action == "remove" &&
              owner == LKEY)) &&
            element_class_name == "pkt_counter" &&
            ((domain == "user" &&
              sha1sum == "704a5c879cb4711f9a55") ||
             (domain == "kernel" &&
              sha1sum == "68a45f3209b3c4c7f621"))
            -> "true";
Signature: "sig-rsa-sha1-base64:FAKE_SIGN_AUTHORIZER"
```

Fig. 5. Example of a Corral credential

name, host, domain and port number of the source or destination elements. All of the properties can be tested. Together they provide fine-grained admission control. An example of a credential with mock keys, checksums and signature is shown in Figure 5. The static Corral admission control is complementary to the even finer-grained static and dynamic resource control of the *OKE*.

#### *3.2.4 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.

### *3.3 The Active network*

The AN runtime used in the example setup is derived from a home-grown active network, capable of supporting either a Java or 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 (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 for receiving the next packet. 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.

### *3.3.1 AN and channels*

The runtime back-end was modified to sit on top of the *OKE Corral*. 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 Corral 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 containing few runtime checks.

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 a 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.

### *3.3.2 Discussion*

When loading unknown new components in the data-path on behalf of third parties, 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.

## *3.4 The IXP1200 network processor*

We built a prototype implementation of the Corral on the Intel IXP1200 network processor which will be described presently. We stress that, while functional, this code is still very much 'proof-of-concept' and not geared for speed.

The IXP1200 contains on-chip one StrongARM control processor, and six independent data processors called *microengines* clocked at 200-232 MHz. In addition, each of the microengines has four hardware contexts, a dedicated instruction store of 1K instructions and a fairly large set of registers. The IXP1200 has 4KB of on-chip scratch memory that is shared by all microengines and the board that was used in these experiments contains off-chip 8MB of SRAM and 256MB of SDRAM.

In previous work, we have described how to provide the OKE mechanisms on the IXP1200 and we will not repeat that explanation here [BS03]. Instead, we sketch how the Corral elements are implemented. Each microengine corresponds to one CORRAL element. The StrongARM is responsible for starting, stopping, and loading individual microengines (and thus the corral elements).

Unlike with its successor the IXP2400, a microengine on the IXP1200 has very limited capabilities of communicating with its neighbouring microengines. As a result, part of the 4KB of scratch memory is used to keep track of connections between the different elements in a simple array. Because an element cannot call a function from another element directly (as it resides on a different microengine) it writes a message in scratch, in the space corresponding to this connection, and sends a signal to the target element. The sending element waits until the receiving element removes the message. Every connection holds space for one message at any time. The implementation details are hidden from

the users, who issue `push()` and `pull()` operations, exactly like they would do on the host processor.

Because the StrongARM can read and write scratch memory directly, it is used to connect and disconnect the elements residing on the microengines. Additionally, it can directly write to the message passing space, thus allowing for elements on the StrongARM (either in the kernel or in userspace) to send data to elements on the microengines. Since the microengines cannot directly pass signals to the StrongARM, sending data from a microengine element to a StrongARM element can only be accomplished by having the StrongARM side monitor the relevant message passing space in scratch memory.

The IXP and the host computer communicate by sharing memory, and by sending messages over the PCI bus. While it is fairly simple to use the doorbell interrupt for this purpose, as an initial solution we have used shared memory communication with explicit polling on both sides (StrongARM and host processor). As a result, it is not currently possible to push or pull data to or from the microengines directly from the host, bypassing the StrongARM. The connections between StrongARM and host computer work analogous to those between StrongARM and microengines, i.e., the host computer polls the relevant space in memory to obtain messages passed by the StrongARM.

To facilitate use of the IXP1200, we implemented the elements as templates that can be easily extended by users who need not worry about complex matters such as pushing and polling via scratch memory, etc. All the complexity is hidden behind boilerplate code provided by the Corral framework. By adding code to a template new classes can be created that can subsequently be loaded and instantiated like any other class.

The implementation of the OKE Corral in microengines, kernel and userspace later inspired the development of FFPF [BdBC+04]. FFPF has no notion of operations like `push` and `pull` that can be executed by the elements, but implements a fairly static dataflow machine instead. On the other hand, it fixes many of the problems that were left as loose ends in the Corral implementation on the IXP (e.g., it reduces copying, offers various communication channels across the PCI bus, etc.).

## 4 Results

We do not 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 [ST93,SOK01][2]. 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 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 [BS02a].

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 6. 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.

We evaluate 3 different implementations: (A) all components in *OKE*, (B) all

[2] The NAPI patch for Linux turns an interrupt-driven kernel into a polling one, whenever the load goes up.

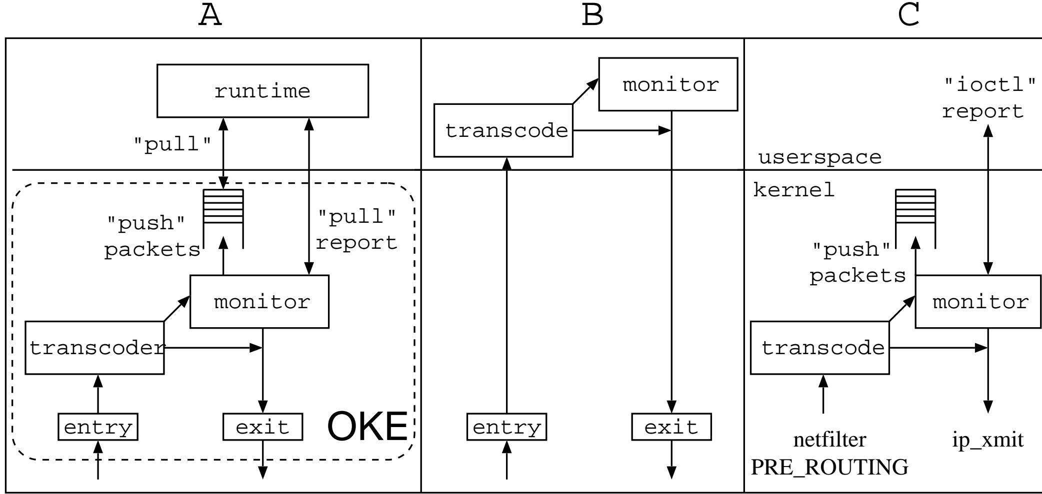
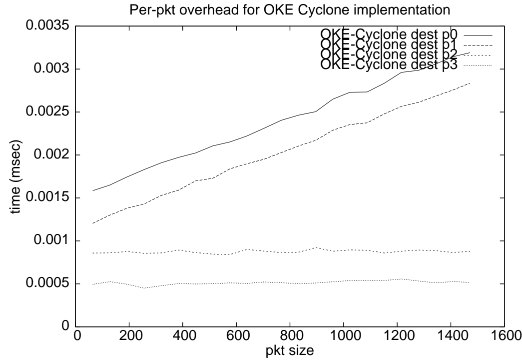



Fig. 6. The three implementations of the applications

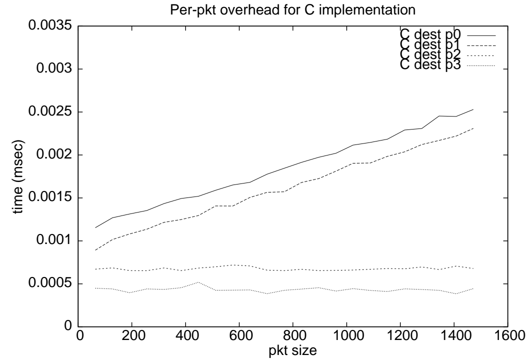
components in AN runtime, and (C) all Corral components in in-kernel C, as shown in Figure 6. All three versions are possible in the *OKE Corral*, but we are most interested in solution (A), as it provides maximum flexibility while still running natively in the kernel. We measure time between packet 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 7(a)-7(c). 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 a 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 fairly inefficient) context switching.

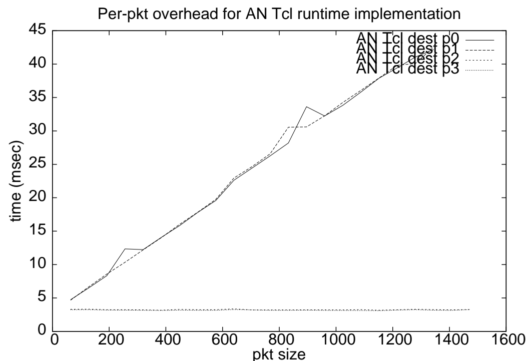
For none of these solutions have we attempted any manual optimisation. Moreover, it is clear that there exist much faster AN runtimes than the Tcl environment that we have used. However, in previous work we measured that a copy from kernel to userspace using a direct `ioctl` pipe to the kernel takes approximately 2 $\mu$s in the best possible case, and considerably longer if we use `libipq` (we measured 8 $\mu$s on average for a copy from kernel to userspace). If a copy to userspace is needed, it will be difficult 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 4 $\mu$s, just on the copies. As shown in Figure 7(a), this overhead alone exceeds the total time needed by the OKE-Cyclone implementation. Even so, we are currently investigating faster userspace implementations. The copy overhead can be avoided if queues are memory mapped to userspace. Even in this case, the cost of context switching remains. Moreover, coordination between kernel and userspace becomes more



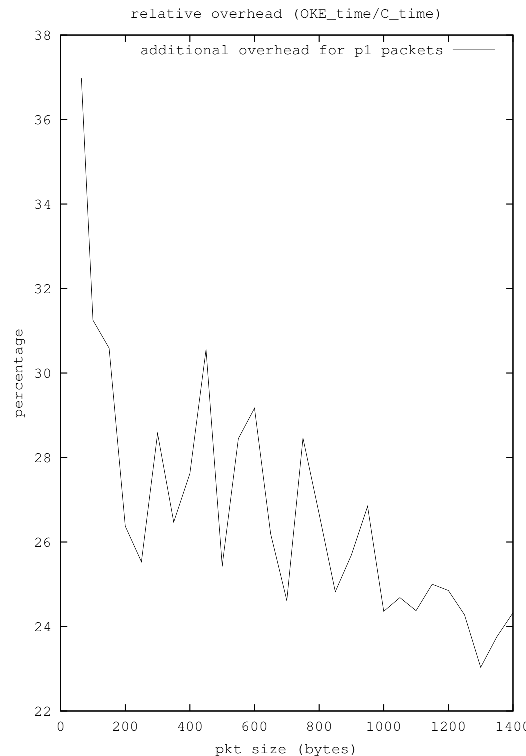

(a) Scenario A: OKE-Cyclone in kernel




(c) Scenario C: native C in kernel




(b) Scenario B: AN runtime in userspace




(d) Overhead of scenario (A) compared to scenario (C)

Fig. 7. Results for the three scenarios: (a) all in the kernel in OKE-Cyclone, (b) all in userspace AN runtime, (c) all in the kernel in C, and (d) overhead of processing packet $p_1$ in the OKE compared with native C

complex. In the network monitoring framework known as FFPF, we have taken exactly this approach [BdBC+04]. However, as far as coordination is concerned (pure) monitoring is much simpler than what is done in the Corral, as packets only travel 'up' and are never injected into the lower levels again.

In Figure 7(d) 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

Table 1
Overhead of pushing to and pulling from the microengines on the IXP1200

| **Operation** | **Cycles** |
|---|---|
| Push to empty element | 53 |
| Pull from empty element | 59 |
| Push to element with 4B data transfer to buffer and simple counter | 71 |

Figures 7(a) and 7(c). 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 ammortised 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. Given the appropriate credentials it may even load new (but known to be safe) element classes.

As shown in Table 1, pushing to and pulling from microengines on the 232MHz IXP1200 takes several tens of cycles. Whether this overhead is acceptable depends on the rate at which these elements process packets and how often they need to push or pull. For instance, for network processing at 1Gbps and minimum sized packets we have roughly 120 cycles available per packet per microengine (significantly more for larger packets). If the load is shared by a number of microengines (and indeed by a number of threads) these speeds allow sufficient cycles for significant work to be done by the elements on the microengines. Still, we stress that so far the Corral on IXP implementation is only a proof-of-concept to see whether we can make network processors fit in the easy-to-use corral framework.

## 5 Related work

Organising AN software in a hierarchical fashion is advocated in many active network projects, e.g., SwitchWare [AHK+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+99]. A third aspect, the separation of control and data path in programmable networks has also been advocated in other projects, e.g., SwitchWare at UPenn [AHK+98]. The router plugins developed at the University of Washington [JJH02] provided the ability to load code dynamically, but did not address the safety issue.

Running code in the kernel of an operating system is fairly common in packet filtering. The original packet filter provided support for a stack-based filtering language [MRA87]. McCanne and Van Jacobson argued that stack-based languages had inferior performance on modern processor compared to register-based language and developed the BSD Packet Filter which has become one of the best-known packet filter approaches in the kernel. A more recent project, known as FFPF, implements a framework that is language neutral and explicitly addresses the safety problem [BdBC+04]. In the current implementation it supports BPF, C functions and a new packet processing language known as FPL-2. Moreover, like the Corral it is capable of pushing processing elements to an IXP1200 network processor [CdBB05]. However, memory handling in FFPF is more efficient than in the Corral. For instance, there is no more copying between kernel and userspace.

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[WLAG93], as well as OS-based approaches such as Nemesis [LMB+96], ExoKernels [EKO94], and SPIN [BSP+95]. Trust management combined with module thinning in ANs was introduced in the Secure Active Network Environment [AHK+98]. A three level architecture that resembles that of the OKE Corral is proposed both by Alexander and Smith in [AS99] and by the first author in [Bos99]. 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 while it 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 [BS02a].

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

Table 2
*OKE Corral* features mentioned in the text compared with other systems. Symbols: '+' = strong support, '–' = weaker, '+/–' = partly, '0' = not applicable to this system.

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

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 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 2. 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.

The OKE Corral was first described in [BS02b]. Our processing hierarchy resembles that of the 'extensible router' [SPB+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, supports kernel plugins with explicit signalling for plugin installation [KRGP02]. 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. More recent than the OKE Corral is RBClick from the University of Utah which takes an approach similar to the OKE [PL03] and also builds on Click and active networking technology. It differs in various important ways from the *OKE Corral*. For example, in RBClick, the language itself is severely restricted (e.g., no normal loop constructs). Also, there is no full interaction with the rest of the kernel as found in the OKE, and it is not implemented on a commonly used OS..

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 also not a commonly used OS.

Early work on the use of Cyclone for kernel work and KeyNote for policy control was demonstrated in FLAME [AIM+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.

## 6 Conclusions

In this paper, we have presented the *OKE Corral*, an environment that allows users to develop fast packet processing architectures. It builds on a set of technologies recently developed in the research community to achieve high-speed programmable packet processing. The Corral, 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 slow-speed code at runtime. While the *OKE Corral* is generic and can be used in many target domains, we developed a high-speed active node (ANode) to demonstrate its usefulness. In contrast with most ANode implementations, the active code can be loaded anywhere in the processing hierarchy, from the runtime to the kernel (and even network processors). 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 ANode that was implemented on top 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 ANode 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 heterogeneity, as well as 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 and Pieter Simoons for significantly improving the Corral code.


## References


[AHK+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+02] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, Michael B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proceedings of IFIP/IEEE Network Operations and Management Symposium (NOMS) 2002*, April 2002.

[AS99] Scott Alexander and Jonathan Smith. The architecture of ALIEN. In *Proceedings of IWAN'99*, pages 1–12, Berlin, July 1999. Springer-Verlag.

[BdBC+04] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[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.

[Bos99] Herbert Bos. *Elastic Network Control (Ph.D.thesis).* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, U.K., August 1999. Also available as Technical Report TR 483.

[BS02a] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *Proceedings of OPENARCH'02*, New York, USA, June 2002.

[BS02b] Herbert Bos and Bart Samwel. The OKE Corral: Code Organisation and Reconfiguration at Runtime using Active Linking. In *Proceedings of IWAN'2002*, Zuerich, Switzerland, December 2002.

[BS03] Herbert Bos and Bart Samwel. HOKES/POKES: Lightweight resource sharing. In *Proc. of ACM SIGBED's Third International Conference for Embedded Software (EmSoft'03)*, pages 51–66, Philladelphia, PA, USA, October 2003.

[BSP+95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, 1995.

[CdBB05] Mihai Cristea, Willem de Bruijn, and Herbert Bos. FPL-3: towards language support for distributed packet processing. In *Proceedings of IFIP Networking*, Waterloo, Ontario, Canada, May 2005.

[CM01] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of the 2001 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, 1999.

[EKO94] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The exokernel approach to extensibility (panel statement). In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (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`.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, 1991.

[JJH02] J.S.Turner, J.W.Lockwood, and E.L. Horta. Dynamic hardware plugins (dhp): exploiting hardware for high-performance programmable routers. *Computer Networks*, 38(3):295–310, February 2002. plugins in Xilinx FPGA.

[JMG+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.

[KRGP02] Ralph Keller, Lukas Ruf, Amir Guindehi, and Bernhard Plattner. Promethos: A dynamically extensible router architecture for active networks. In *IWAN 2002 Fourth Annual International Working Conference on Active Networks*, ETH Zurich, Switzerland, December 2002. Springer.

[LMB+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. *IEEE Journal on Selected Areas In Communications*, 14(7), September 1996.

[MBC+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.

[MRA87] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th Symposium on Operating System Principles*, pages 39–51, Austin, Tx., November 1987. ACM.

[NL96] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, Seattle, Washington, October 1996.

[PL03] Parveen Patel and Jay Lepreau. Hybrid resource control of active extensions. In *OPENARCH'03*, San Francisco, CA, April 2003.

[Rit84] D. M. Ritchie. A stream input-output system. *AT&T Bell Labs Technical Journal*, 63(8):1897–1910, 1984.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *USENIX*, November 2001.

[SPB+02] Nadia Shalaby, Larry Peterson, Andy Bavier, Yitzchak Gottlieb abd Scott Karlin, Aki Nakao, Xioahu Qie, Tammo Spalink, and Mike Wawrzoniak. Extensible routers for active networks. Princeton work. Probably published somewhere., 2002.

[ST93] Jonathan M. Smith and C. Brendan S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, 1993.

[WLAG93] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault-isolation. In *Fourteenth ACM Symposium on operating System Principles*, pages 203–216, December 1993.