The OKE Corral: Code Organisation and Reconfiguration at Runtime Using Active Linking

Herbert Bos and Bart Samwel

LIACS, Leiden University, Niels Bohrweg 1, 2333 CA, The Netherlands
{herbertb,bsamwel}@liacs.nl

Abstract. The OKE Corral is an active network environment which allows third-party active code to configure an active node's code organisation at any level, including the kernel. Using the safety properties of an open kernel environment and a simple 'Click-like' software model, third parties are able to load native code anywhere in the processing hierarchy and connect it to existing components at runtime.

1 Introduction

For reasons of safety, most active networks (ANs) tend to sandbox active code in user space, either locally or at a remote node. Moreover, such code is often interpreted, which slows down its performance considerably. Even in non-active environments interpreters are frequently used whenever application-specific code is loaded in the kernel. A well-known example is found in BSD packet filtering.

In previous work, however, we have shown how the open kernel environment (*OKE*) provides a safe, resource-controlled programming environment which allows fully optimised native code to be loaded in a Linux kernel by parties other than `root` in a safe manner [BS02]. In this paper, we describe how the *OKE* was used to develop an environment for building high-speed ANs allowing third parties to load and configure native code anywhere in the processing hierarchy. An implementation of this environment is found in the *Corral* (Code Organisation and Reconfiguration at Runtime using Active Linking). The contribution of this work is that three existing technologies in programmable networks (open kernels, the 'Click router' model, and ANs) are combined to provide a safe platform for fast packet processing in the kernel of a common operating system while explicitly separating control and data. In the *OKE Corral* high-speed packet processing is managed by slow-speed control code. It can be summed up as follows:

1. We borrow the LEGO-like software model advocated by the 'Click router' project in [CM01] to build both fast *data* paths and slower *control* paths.
2. One of the components on the control path is an AN runtime.
3. The configuration/implementation of the paths is controlled by third-party code executing either on the node itself, e.g. in the form of active applications (AAs), or at a remote site.
4. The *OKE* ensures that new kernel-level path components are safe.

J. Sterbenz et al. (Eds.): IWAN 2002, LNCS 2546, pp. 32–47, 2002.

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,

Not all issues concerning the use of the *OKE* for ANs are addressed in this paper. In particular, node heterogeneity and scaling of trust relationships to large networks are not addressed. However, while the *OKE* relies on trusted compilers, the issue of trusting compilers in a remote domain is non-trivial and important. We will briefly discuss this in Section 3.3.

The remainder of this paper is organised as follows. The *OKE Corral* architecture is explained in Section 2, the prototype implementation of the architecture is discussed in Section 3. This is evaluated in Section 4. Related work can be found in Section 5 and conclusions are drawn in Section 6.

2 Architecture

As illustrated in Figure 1, the *OKE Corral* builds on three technologies: (1) the *OKE*, (2) one or more AN runtimes, and (3) packet channels that implement control and data paths. 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).

2.1 Corral Terminology

The terminology in the *OKE Corral* roughly follows that of the Click-router project, although there are some differences. What are called processing elements in Click are called 'engines' in the *Corral*. They may have multiple input and output ports that can be logically attached to other elements to form connections (drawn as arrows). 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*. Connections are 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 and pushes it to a destination, changing the pull into a push.

The inverse of a pull2push element is a queue, as it accepts pushes on its input, and pulls on its output, and thus may be termed a *push2pull* element. 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

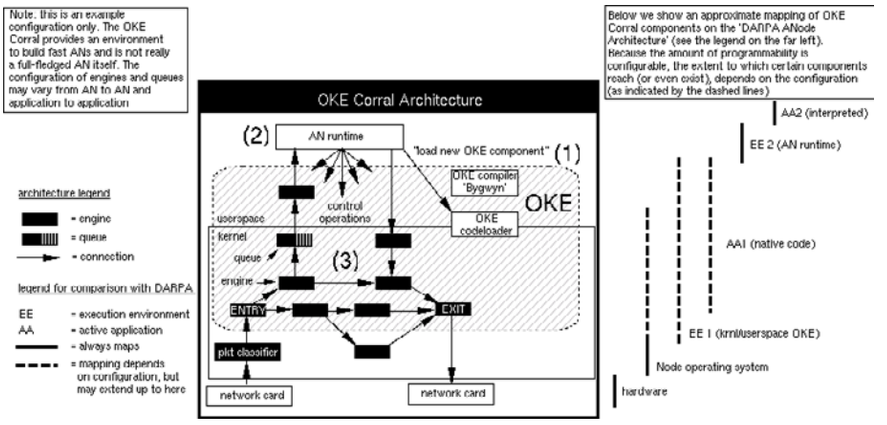

Fig. 1. Overview of the *OKE Corral*

path followed by a particular packet is known as a 'channel'. In Figure 1, the entry engine, the two boxes to the right of it, and the exit engine form a channel.

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

2.2 OKE, AN Runtime, and Channel Interaction

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

The AN is allowed to disconnect the two engines and reconnect them (at runtime), e.g. to new components inserted between them. For example, to receive all packets in its runtime, a trivial AN implementation might: (1) disconnect the two engines, (2) reconnect the entry engine to a queue, (3) implement an engine's interface for the runtime (essentially making the runtime an engine 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. All incoming packets classified as AN traffic are now 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). These stan-

dard components can be highly optimised so as to incur few checks at runtime. In addition, the AN is able to load entirely *new* components. In case native code is to be loaded in the kernel, the *OKE* is used to ensure safety. As discussed in Section 3.1, the *OKE* is able to restrict code according to the loading party's credentials. In *OKE* terminology, the credentials presented by a client's are defined as that client's *role*. Thus, a party in a highly untrusted role may be allowed to load code with very few privileges and many dynamic checks, while a party in a highly trusted role may benefit from a more relaxed security policy.

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 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 containing the control code. Given the appropriate privileges they are able to replumb, or add new elements to, the data-path at runtime. Thus, the amount of data-path programmability allowed is configurable, which is useful if active networks are to scale.

3 Implementation

3.1 The Open Kernel Environment

Although the running of third-party code in the kernel normally violates security constraints, it would be useful from a performance perspective. 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* determines the privileges of user and code, both at compile time and at load time. Based on these privileges a *trusted compiler* enforces extra constraints on the code. In the following we briefly describe the *OKE*'s two main components: the code loader (CL), and the *bygwyn* compiler (as previously presented in [BS02]). A pre-release of the *OKE* as well as an extended version of this paper can be downloaded from `www.liacs.nl/~herbertb/projects/oke/`.

The CL accepts object code, together with authentication and credentials, from any party. It checks the credentials against the code and the security policy and loads the code if they match (Figure 3). Trust management is based on KeyNote [BFIK99]. At start-up, the CL loads a security policy, containing the keys of any party permitted to load certain types of modules. These parties are then able to delegate trust to other clients by way of credentials containing containing the 'rights' that are granted, e.g. the right to 'load modules of *type* X or Y, but only under *condition* Z'. A 'type' here denotes the set of privileges given to the code, e.g., the interface to the kernel, the amount of CPU time, etc. The 'condition' contains environment-specific stipulations, e.g., 'only valid during

office hours'. A module type is instantiated when source code corresponding to it is compiled. The trusted compiler generates an unforgeable 'compilation record' which proves that module M was compiled as type T by this compiler.

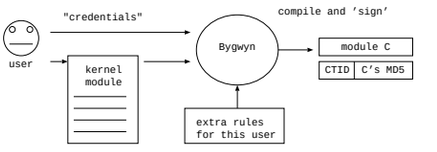

Fig. 2. User compiles kernel module

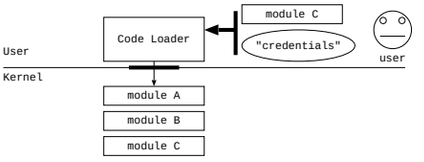

Fig. 3. User loads module in the kernel

It is crucial that we guard against malicious or buggy code 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 interpreter, as such a language necessarily restricts towards the lowest common denominator. Instead we would like to have a single language that is automatically restricted on the basis of explicit privileges. A single language is preferable to many special-purpose languages for many reasons, e.g., consistency, learnability, maintainability, flexibility, etc. Moreover, using a language like C would facilitate the interfacing of third party code to the rest of the kernel.

We therefore allowed a C-like 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 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+02]. However, for true safety and speed, using Cyclone was not sufficient. For example, we had to add an entirely new garbage collector to deal with pointers to kernel memory. Other hard problems (e.g., resource limitation, module termination, and memory/pointers sharing) are also not solved by Cyclone. We therefore created our own dialect which we call 'OKE-Cyclone'.

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, **b**ut **y**ou **g**et **w**hat **y**ou **n**eed'). *Bygwyn* is customisable, so that in addition to its normal language rules, it is able to apply extra rules as well. 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.

The key idea is that the customisations for a user's program depend on the user's role: users present credentials to the compiler, and these credentials determine which rules are applied (Figure 2). 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. 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.

Depending on the users' roles, they get access to the rest of the kernel via an API containing the 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 its role. In other words, the API is used to *encapsulate* the rest of the kernel (Figure 4). In the figure, some function calls are relegated to a wrapper, while others may be called directly.

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 global code analysis to decrease the number of dynamic checks.
2. Environment setup code (ESC) containing 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 them using code declared by the ESC.
3. Certain language constructs can be removed from the programmer's repertoire using a new `forbid` construct (examples include: `forbid extern "C"`, `forbid namespace`, and `forbid catch`).
4. A unique, randomly generated namespace is opened to prevent namespace clashes and unauthorised imports of symbols from other namespaces.
5. The stack usage of the code can be restricted to a limit defined in the ESC.
6. CPU usage is limited by using a modified timer interrupt. When a module has not finished on time, an exception is thrown and the module is removed. Code misbehaving in other ways is likewise removed.
7. Cyclone's region-based memory protection mechanism was extended with a new region `` `kernel ``, to distinguish between kernel-owned and module-owned memory regions and a new garbage collector was implemented to ensure that pointers from the *OKE* modules to kernel memory (which may be manipulated by kernel functions) are memory safe, and freeing of module memory is handled correctly.
8. Specific fields of kernel structures shared with untrusted code can be statically protected by making making them `locked`. A `locked` member cannot be used in calculations, it cannot be cast to another type, no other type can be cast to it, no pointer dereferences can take place, and no structure members can be read. Basically, its is limited to copying, and it cannot be read. This technique reduces the need to anonymise data at run-time.

3.2 Channels

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 *OKE* channel elements all have simple interfaces that are implemented in either C or OKE-Cyclone. 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 the main features of engines and queues. In essence, queues and engines have unique identifiers and communicate by pulling and pushing data from and to each other's ports. A push or pull connection is typed, so that only specific items may be pushed or pulled on a connection. The types range from simple types such as integers and octets to composite types (e.g., IP packets). We have not addressed the issue of how specific engines or queues to connect to are discovered or located. This should not be a problem for a handful of elements that we loaded ourselves on a single node, but for large-scale deployment such functionality would be very useful.

Queues in the default implementation are strictly FIFO (producer/consumer 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 never initiate actions themselves. In contrast, engines are active elements. Apart from push and pull, they also provide a control interface (Figure 1), and a `run` method which is called when it is scheduled. The control interface contains the `connect` methods needed to link engines to other engines or to queues. These methods take as arguments (among other things) the unique name of the target element, as well as the port and the port direction (input or output). Queues do not provide such methods: they are managed by engines.

Engines and queues can be (dis-)connected 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 replumb an element when it is active (e.g., about to push a packet to an engine we would like to disconnect), these activities are protected by a 'readers-writers' solution: many different data-path actions may be taking place at any time, but management operations such as `connect` require exclusive access.

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, or complex types by passing pointers, making communication within the same domain quite efficient. It is also possible to place engines and queues in different domains. For this purpose we use simple marshalling techniques commonly used in remote procedure calls. For example, if an engine in domain D_1 wants to push a 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. 'Remote' here means a different domain, which could easily reside on the same host. Default proxies and

marshalling routines have been written which are expected to suffice for most applications. Even so, the scheme can be easily extended.

Packet traversal in the *Corral* is as follows. Once a packet is classified (by the classifier in Figure 1) as belonging to the AN, it is pushed on the AN's entry engine and follows the data-path determined by the AN's engines and queues. Some of the fields in the packet may be protected against access violations using the `locked` keyword. 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 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 runtime provides a simple environment for AN experiments and permits code loading both in-band and out-of-band. It consists of an interpreter and a fairly extensive set of operations specific to the AN. This is called the *core set*, which is implemented in C. The core set contains elementary operations, e.g. functions to access received packets and to find the load on specific links, etc. It also contains a `send` operation for transmitting a packet. 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. A 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. An additional library that is fully implemented in Tcl contains a large number of functions that are commonly used, as well as wrappers around the core set.

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, the active code in the runtime 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 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 such modules can be 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 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 refers to new kernel modules on a remote

webserver. Similarly, it uses URLs to refer to the credentials. Next, the module and the credentials are both loaded and offered to the *OKE* codeloader. Provided the credentials match, the module is pushed into the kernel. At that point the AN is able to manage the new engines and queues in exactly the same way as the pre-installed components.

When loading new components in the data-path, safety is guaranteed by the *OKE*. This means not only that the code *must* be written in OKE-Cyclone, but also that the compiler that compiles it must be *trusted*. We have not addressed the issue of whether 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 are accepted by many sites (the "VeriSign model"). Alternatively, we might 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 a module in the kernel it is required to authenticate every such request by signing a number of items with its private key. However, if code is running on an unknown active node, clients may be reluctant to send their private keys there for obvious reasons. In the *OKE Corral* model, we have 'single sign-on' behaviour. In other words, the identity of the client is established when 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.

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, such numbers often say more about the traffic *capture* (e.g. polling or interrupt-driven) than about the *processing* [ST93]. Second, we are really interested in how the *Corral* compares to typical ANs and this concerns primarily the nature of code execution: in-kernel native code versus interpreted code in userspace. For the number of packets per second that *can* be processed with a channel-based system, please refer to [CM01].

Instead, we measure the performance of the data-path components and compare the results with alternative implementations. All measurements 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 roughly 250 nsecs (including all locks and sanity checks). The applications used for the comparison are in the domains of transcoding (application T) and monitoring (application M). Both are considered components on the data-path. In the *OKE Corral* version of the experiment, they are implemented in OKE-Cyclone, and 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 is read by the AN monitoring

application in userspace using a pull. On request (pull), M also reports 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 IP and UDP checksums. 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.

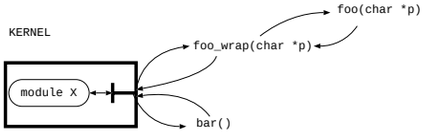

Fig. 4. Kernel encapsulation

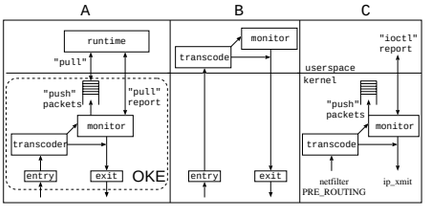

Fig. 5. The three scenarios used

We evaluate 3 different implementations: (A) all components in the *OKE*, (B) all components in the 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, since the entire payload must be processed for T, the overheads for p_0 and p_1 packets strongly depend on the packet sizes. The p_2 and p_3 graphs on the other hand are basically flat, as we do not even need to recalculate the checksums for these packets. We also observer that in the Tcl implementation the effect of monitoring is no longer visible. This is due to the enormous overhead introduced by the interpreter and context switching.

No manual optimisation was used in any of the implementations. Moreover, there exist much faster AN runtimes than the one we have used. However, in previous work we measured that a copy from kernel to userspace using an `ioctl` channel takes roughly 2 μs, and considerably longer with `libipq` (8 μs on average). 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 4μs, just on the copies.

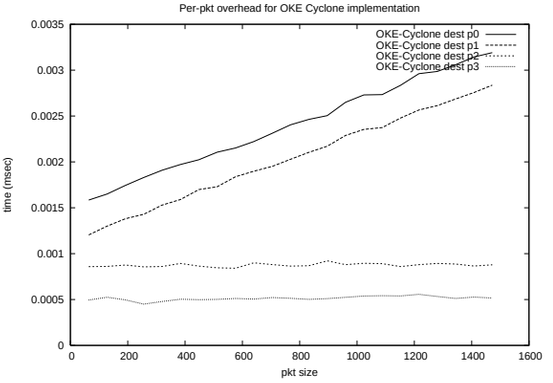

Fig. 6. Scenario A: in-kernel in-Cyclone

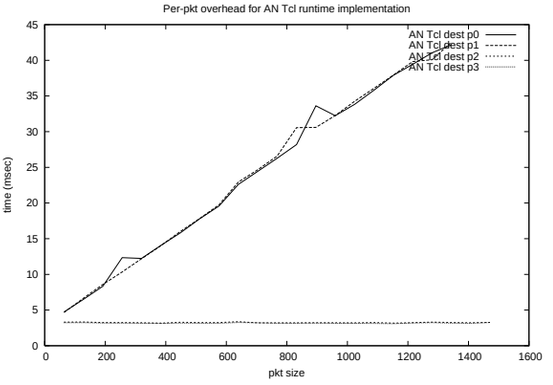

Fig. 7. Scenario B: in-userspace (AN)

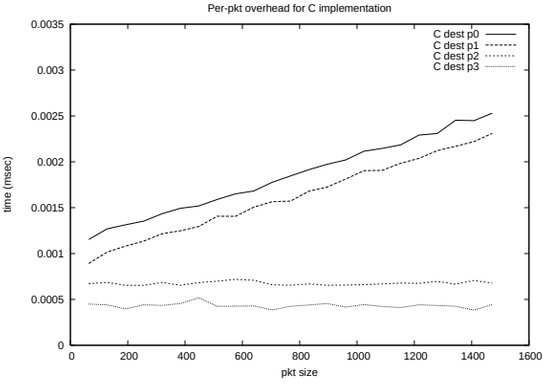

Fig. 8. Scenario C: in-kernel in-C

This overhead alone exceeds the total time needed by the OKE-Cyclone implementation.

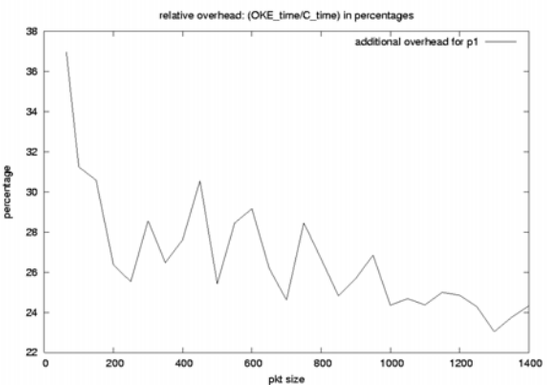

Fig. 9. Overhead of processing packet p_1 in the OKE compared with C

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 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 roughly 25%. A substantial gain in performance can be achieved by employing the *OKE* in ANs. However, even 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.

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 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+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]. 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 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 are shown in Table 1. Below we discuss the projects mentioned in the table.

We have been strongly influenced by the Click router's LEGO-like organisation of forwarding code [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+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

	Feature	SILK	ANN	PromethOS	SPIN	FLAME	Click	OKE Corral
1	3rd party code in krnl	++	−−	−	++	++	−−	++
2	full krnl programmability + restriction possibilities	+/−	+/−	+	++	−	+/−	++
3	resource control	+	−	−	+	+	−	++
4	safety guarantees	−−	−	−	++	+	−−	++
5	LEGO-like components	+	−	−	−	−	++	++
6	dynamic configuration	++	+	+	++	+	−	++
7	separation control/data	++	−	++	0	+	+	++
8	AA: capsules load in krnl	−	−	−	0	0	0	++
9	AA: out-of-band loading	++	+	++	0	++	+	++
10	common OS	++	++	++	−−	++	++	++

Table 1. *OKE Corral* features compared with other systems. Explanation of symbols: '+' = strong support, '−' = weaker, '+/−' = partly, '0' = not applicable to this system.

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*.

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+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

The *OKE Corral* combines a 'Click-like' software model with an open kernel environment under control of an active network while maintaining strict separation of control and data plane. Performance varies with the programmability desired. At one extreme, only the control plane is programmable, while data-paths are composed of highly optimised 'standard' components. At the other extreme, the 'capsule' approach can be supported. In between these two extremes, but closer to the former, we have the *OKE* channels. For flexibility, the different kinds of programmability may be mixed, so that capsules, pre-defined and third-party components all interact to build data and control flows.

Acknowledgements

We are indebted to Lennert Buytenhek for his help on the thornier issues in kernel hacking and Nadia Shalaby and the reviewers for their excellent feedback.

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 *Proc. of NOMS'02*, April 2002.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A.D. Keromytis. The KeyNote trust-management system version 2. *NWG 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+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.

[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.

[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.

[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. *JSAC*, 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.

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

[NLA+02] N.Shalaby, L.Peterson, A.Bavier, Y.Gottlieb abd S.Karlin, A.Nakao, X.Qie, T.Spalink, and M.Wawrzoniak. Extensible routers for active networks. In *DARPA AN Conference and Exposition*, June 2002.

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

[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.

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