Safe Kernel Programming in the OKE

Herbert Bos and Bart Samwel
Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Email: {herbertb,bsamwel}@liacs.nl
Phone: +31-71-527 7033, Fax: +31-71-527 6985

***Abstract*— This paper describes the implementation of the *OKE*, which allows users other than root to load native and fully optimised code in the Linux kernel. Safety is guaranteed by trust management, language customisation and a trusted compiler. By coupling trust management with the compiler, the *OKE* is able to vary the level of restrictions on the code running in the kernel, depending on the programmer's privileges. Static sandboxing is used as much as possible to check adherence to the security policies at compile time.**

I. Introduction

THE open kernel environment (*OKE*) allows users other than the root to load native and fully optimised code in the kernel of a UNIX-based operating system [BS01]. In this document, we explain both the *OKE*'s implementation and the way in which programmers use it to build and load safe kernel modules. To illustrate the programming model, we discuss example applications that perform packet transcoding.

The premise for this paper is that it is useful to allow third parties to add functionality to the kernel. Generally, a modern operating system (OS) allows only root-users to do so, and all software needed for such tasks as network monitoring is either installed by the system-administrator, or run in user space. Allowing third-party code into the kernel jeopardises all security constraints, as the code can 'do anything'. For untrusted code this is not acceptable. From a performance perspective, on the other hand, it would be useful. For example, time-critical network applications, such as transcoders, monitors and filters, want to avoid crossing the kernel-userspace boundary for every packet, because of the overhead. Instead of relegating flows of packets to userspace for processing, one would rather do as much processing as possible in kernel space and send only the results to the application. Moreover, one would rather use fully optimised native code than an interpreted filter language such as used by BSD packet filters.

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.

The remainder of this paper is organised as follows: related work is discussed in Section II, the architecture of the *OKE* in Section III, example applications in Section IV, and conclusions in Section V. Implementation details will be given throughout the text.

II. Related work

A common solution for permitting foreign code to execute safely in a security-sensitive system is to run it inside a safe execution environment, such as a virtual machine. The prime example (in user space) is Java [GJS96]. Unfortunately, Java relies on many run-time safety-checks, e.g., for array bounds. These checks are the same regardless of whether the code loading party is trusted or not, and *always* incur a serious performance penalty. Moreover, due to the open nature of Java (e.g., the capability of loading new classes at run-time), it is often not possible to safely optimise away these checks. Other interpreted solutions also suffer from bad performance, as is witnessed by the well-known BSD Packet Filters, which are implemented in an interpreted stack language and execute in the kernel [MJ93].

The need for clear delineation between some notion of 'kernel' code and 'application' code, even in a Java virtual machine, to provide isolation and resource control was argued by [BHL00]. Kernel boundaries serve as safe 'termination boundaries', so that termination is deferred if code is running in the kernel. In the *OKE* we recognise the importance of safe termination, as discussed in Section III-

D. The DrScheme programming environment offers support for multiple language levels in Scheme [FFF+97], so that some programmers get access to only a small subset of the language, while others are allowed to use the more complex constructs. The Cyclone extensions in the *OKE* also offer customisability of the language (controlled by trust management), but with a focus on running a fast programming language, familiar to systems programmers, in the kernel of an OS without jeopardising security constraints.

The *OKE* also differs from either of these approaches in that it allows users to load native code, optimised statically for maximum performance. It is related to recent work in software fault isolation (SFI), proof carrying code (PCC), language restriction, extensible compilers, and operating systems such as SPIN, Nemesis, and the ExoKernel.

SFI, advocated by [WLAG93], can check memory accesses with a runtime overhead that can be reasonably small, depending on the application and the form of SFI used. However, memory access is just a small part of the full spectrum of security issues, and the *OKE* therefore allows for a much wider range of restrictions.

PCC achieves safety through the means of proof generation and checking [NL96]. In PCC, the kernel publishes a *safety policy*, specifying formally what code is acceptable. The code loading party should provide a formal proof that the code adheres to this policy. PCC achieves provable code safety and protects against proof or code tampering. However, because of this, it is rather complex. Generating a proper safety policy which 'covers all bases' is difficult and so is the generation of proofs for the kernel modules. For simple problems, some benefit may be expected from automated proof generation, but as the complexity rises, this becomes more difficult. The *OKE* takes what may be considered an 'engineering' approach to the problem of safety in the kernel. Users are prevented from generating or loading dangerous code by way of trust management. Their code is explicitly given access to parts of the kernel on the basis of the credentials provided for it. Because of the need for trusted components (e.g., compilers) in the architecture, however, this may make the *OKE* less suitable than PCC for use in active networks. We will return to this issue in the discussion at the end of this paper.

Language restriction facilities exist in a number of (often interpreted) languages [Bor94]. The idea is that given an interpreter for a language, it is possible to create a new 'safe' interpreter, which provides access to only a small subset of the programming language, guaranteed to be safe. The *OKE*'s language restriction is related to such efforts, but with a greater arsenal of possible restrictions.

Extensible compilers are presented in [ECCH00]. Extensions are written in a special-purpose extension language that is quite good at expressing 'coding conventions' that must be enforced. For example, using the pattern matching capabilities supplied by the language, it is possible to make the compiler scan each function to check that all calls to `cli` (disable interrupts) are followed by a call to `sti` (restore interrupts). The primary application is to find errors of this and other kinds, e.g., in OS code, and not to differentiate the capabilities of programmers based on an authorisation policy.

Allowing users to add code to the OS is possible to some extent in systems like Nemesis, the ExoKernel and SPIN, either because the 'OS software' is not running below a kernel boundary anyway, or because language-specific properties are used to achieve a certain level of safety. These are all completely new, research-oriented operating systems, and not commonly used. In spirit, the *OKE* is closest to SPIN, an operating system that builds on the safety properties of Modula-3 to allow third parties to add kernel code [BSP+95]. We are using some of the same techniques that are found in SPIN. However, the level of safety offered by SPIN is not entirely complete; for example, it does not control the quantity of heap memory used by such 'safe' kernel additions. In the *OKE* we try to pay more attention to these aspects of safety as well. Furthermore, an important distinction between the *OKE* and the other systems is that we look at a very commonly used OS, namely Linux.

Trust management and code loading in active networks were first discussed in [HK99]. A high-level type-safe language (PLAN) was used, to provide restrictions of a packet's service environment based on its level of privilege. Although the *OKE* was one of the first implementations that allowed restrictions (based on privileges) to be placed on a common 'low-level' programming language to permit fully optimised native code to be loaded in the kernel, the first use of the Cyclone programming language for safety, coupled with KeyNote for policy control, as well as additional static and run-time checks was demonstrated in FLAME [AIM+02]. The emphasis of FLAME is on efficient and safe network monitoring and less on making the kernel of an OS fully programmable. *OKE* provides a number of additional features that are needed for general-purpose OS kernel extensions, focusing around the notion of customisable languages. For instance, FLAME provides no flexibility in the amount of restriction that is placed on a kernel module, and there is no concept of allowing full interaction between the module code and kernel (e.g., pointers from Cyclone code to

kernel memory and vice versa). We are currently exploring ways to align the two efforts, e.g., by running FLAME in the *OKE*.

III. Architecture and implementation

The *OKE* architecture consists of three main components: (1) the code loader, which loads a user's code in the kernel, (2) the *bygwyn* compiler, which compiles user's code according to the rules corresponding to the user's privileges, and (3) the *legislator*, which generates these rules. The key issue in the *OKE* is: given that code may be loaded in the kernel, what is it allowed to do there?

A. The Code Loader

The existing Linux code loading facilities (`insmod` and `rmmod`) are extended with a new code loader (CL) which accepts blobs of code, together with authentication and credentials, from any party. The blobs of code are simple kernel module object files, i.e., using a layout based on the standard object file formats. The CL sits between non-privileged users and the normal code loading facilities provided by Linux. Anyone with the right credentials for a blob of 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 if they match, it loads the code in the kernel. The process is illustrated in Figure 1. The user offers the object file to the CL, which checks the user's credentials and if they match the module, the module is pushed into the kernel.

1) Trust management implementation: The delegated trust scheme and authorisation check are implemented using KeyNote [BFIK99] and the OpenSSL library. Trust may be delegated, i.e., an *authoriser* party may delegate part of its privileges to another party (the *licensee*). 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. If required, it is possible to specify in detail who is allowed to load which modules. The CL and 'trusted' clients (who need not reside on the same host as the CL) are then able to delegate trust to other clients.

A trust delegation is encoded in a credential 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 a licensee the right to 'load a module of *type* X or *type* Y, but only under *condition* Z'. A 'type' here denotes the privileges given to the code, e.g., to access certain kernel data structures, to use a certain number of cycles and a certain amount of memory, etc. The 'condition' may contain environment-specific stipulations, e.g., that the right to load these modules is only valid during office hours. The credential is signed by the authoriser, so that it cannot be modified and the protocol uses nonces to ensure that replays are not possible. The client sends its public key to the CL and receives a nonce in reply. In all further exchanges necessary for the request, the nonce is incremented by the client and sent to the CL encrypted with the client's private key.[1] This ensures that the CL is really talking to the client who sent the request and the fact that it is this key for which authorisation is sought (i.e., for which the appropriate credentials should be provided) enables the CL to check whether the client has the required authorisation.[2]

2) Module type instantiation: A module type is instantiated when source code corresponding to the type is compiled, which causes a signed *compilation record* to be created (see also Section III-C.1). The compilation record is itself a credential that links the generated object code with a type. The binding between credential and code is by an MD5 message digest, i.e., credentials refer to kernel modules to be loaded not (just) by their name, but by the code's MD5. This ensures that the CL is able to check that the code presented by the client is indeed the code referred to in the credential. If the code and the credentials match, the code is loaded. In this document, we call the set of credentials used by a client the client's *role*. For example, a particular set of credentials may identify a client's role as 'student loading code in the kernel'.

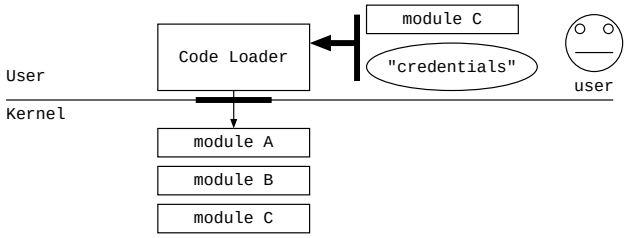

Fig. 1. User loads module in the kernel

B. Elastic languages and APIs

Code that is known to be safe (according to a safety policy) should be allowed to run in the kernel if this improves the performance. Problems arise when code is allowed to use arbitrary pointers, call arbitrary functions, reference arbitrary memory locations, use unrestricted amounts of CPU time (e.g., by running infinite loops, or endless recursion), etc. These are potential dangers that can be spotted

[1] The specific request (e.g., 'load module M') is similarly encrypted.

[2] For the current trust mechanism, see `www.liacs.nl/~herbertb/projects/oke/oketrust.html`.

at the source level, after construction of the abstract syntax tree (AST), or at the level of intermediate code. Once spotted, there are often various ways to solve the problem. For example, possible solutions to prevent programs from consuming too many CPU cycles include:

1) limiting the number of instructions a program may consume by instruction counting at runtime;
2) limiting the number of instructions by worst case estimation at compile time (adding up the worst case instruction count for all branches in the call graph and checking whether it may exceed a maximum);
3) removing all looping and recursion constructs from the language's repertoire (or restricting them such that they can be checked statically);
4) limiting the amount of time spent by the program by stopping the program's execution after a timer expires.

These solutions are orthogonal to the *discovery* of the problem: at some point in the compilation phase we discover that a program has a cycle in the call graph or a branch that leads to a backward jump. We may choose any solution to make the language safe (with respect to this problem) and the tradeoff may be delicate. For example, solution (1) is very flexible but incurs significant runtime overhead, solution (2) is very conservative, solution (3) has no direct performance penalty but leaves us with a rather limited programming language, and solution (4) incurs only little overhead, but is somewhat coarse grained.

What we have tried to avoid in the *OKE* is the definition of yet another safe language which is only useful for implementing filters, say, and/or runs inside a safe interpreted environment. Instead, our aim is to allow one to restrict a normal programming language (preferably something like C), in such a way that, depending on who the code-loading party is, it will get more or less access to resources, APIs and data, and/or incurs more or less runtime overhead.

Our motivation for using code restriction, as opposed to a fixed, safe language is that a fixed, safe language necessarily restricts towards the lowest common denominator. Suppose, for example, that in an academic environment students are prevented from consuming too much CPU time in the kernel, e.g., by the addition of strict runtime cycle counters to the object code. This solution may be fine for students, but not for staff members. Staff members should be given a less restricted language for their own machines, e.g., where CPU time consumption is measured only at infrequent intervals, say once every 30 seconds. A third group, system programmers, may want no restrictions on their CPU usage at all. This already gives us three different wish lists for the language. One option is to use a different language for each class of users, but we feel this is undesirable for many reasons, e.g., consistency, learnability, maintainability, flexibility (what happens if a new class has different requirements?), 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 introduced the notion of *elastic* languages. An elastic language is an ordinary language, on which arbitrary constraints can be placed (serving particular classes of users), either on the availability and use of language constructs, or on the runtime behaviour of the generated code. Conceptually, this means that a *separate* language is created for each group of users, where each language contains a *subset* of the 'mother language' (guaranteeing learnability, maintainability, etc.).

We implemented a fully extensible compiler, capable of enforcing such restrictions, as discussed in [BS01]. Extensions enabled us to completely remove language constructs from the language repertoire, limit their availability (e.g., 'no more than one loop per program'), or even change the compiler's behaviour. In retrospect, the extensions that could be dynamically loaded by the compiler were useful mostly for forbidding access to certain language constructs and for adding runtime checks whenever certain constructs were used. It is rarely useful to change the apparent *behaviour* of the generated code. Moreover, we found that writing full compiler extensions was tedious and often complicated. This is what led us to develop the concept of a *legislator*, a program which is capable of generating compiler extensions of specific types by way of a GUI presenting all sorts of possible extensions.

Since then, we have tried to incorporate customisability in the language itself, for example, by introducing new keywords such as `forbid`. The `forbid` construct allows us to remove constructs from the language (see also Section III-D). Most runtime checks can easily be added by explicitly introducing the concept of *wrapping*, the automatic creation of code wrappers around certain constructs. Extensible compilers are still useful, e.g., to check whether code adheres to coding conventions (see [ECCH00]), but for our purposes, the usefulness is limited. In our current implementation, although the mechanism is still present, we no longer focus on low-level production rules that can be loaded dynamically, concentrating instead on providing the necessary customisability in the language itself. We believe this to be a very promising approach. It will be discussed in Section III-D. As a result, we now have two distinct implementations of the *OKE*'s compiler: an old one, which relies solely on dynamically loadable production rules, and

a new one that incorporates customisability within the language itself. Where appropriate, we will make the distinction explicit.

1) The choice of language: In this section we discuss how the development of the new *OKE* compiler was related to the programming language that was used. To test our ideas, we first implemented a prototype for Pascal, because its grammar is friendlier than C's. It achieved safety by aggressive language restriction and potentially many runtime checks on a somewhat ad-hoc basis. Although we achieved promising results with this version, e.g., in the field of packet filtering [BS01], we realised that at some point we would have to tackle the 'harder problems' anyway and reconsidered the language issue. As system programmers on UNIX-like systems prefer C, our language would ideally be C. This also makes interfacing between our code and the rest of the kernel simpler. However, C is not safe and the possibilities of crashing or corrupting a kernel are endless.

We therefore opted for *Cyclone*, a crash-free language derived from C, that ensures safe use of pointers and arrays, offers fast, region-based memory protection, and inserts some runtime checks, but only when needed [JMG+02]. However, for true safety and speed, we needed both more and less than what is currently 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.

Some really hard problems are also not solved by Cyclone. For example, what to do when a module shares memory with the kernel? Who is allowed to free that memory and when? How do we restrict a module's access to resources? How can the kernel interrupt a module, e.g., because it has run out of resources? For this purpose, we modified the Cyclone compiler, implementing our own solutions to these problems. We will discuss these problems and our solutions in Section III-D. In this paper we will use the term *Cyclone* to refer to both the original language and the modified version we used for this paper. When we discuss features specific to our version we will make this explicit.

C. The Bygwyn compiler

The restrictions discussed in the previous section are enforced by a trusted compiler, known as *bygwyn*, which is based on the observation that was made by the Rolling Stones that 'you can't always get what you want, **b**ut **y**ou **g**et **w**hat **y**ou **n**eed.'

1) Bygwyn implementation: These restrictions required *bygwyn* to provide support for customisability, meaning that in addition to its normal set of rules, it is able to apply extra rules, or customisations, as well. If restriction X is applied, the user code that is compiled is subjected to the compiler's default rules *plus* the additional rules corresponding to restriction X. Both may generate error messages, or change the state of the compiler. For example, we allow one to remove constructs from the language. If after such a restriction the compiler encounters the forbidden construct, it generates an error. In other words, depending on which extra restrictions we impose, the language that is permissible to the compiler may change, and so may the generated code.

The key idea is that, just like code loading, the customisations that are used for a specific user's program depend on the user's role. In other words, users present their credentials to the compiler, and the credentials determine which customisations are appropriate. In this aspect, the *OKE* customisations are coarse-grained, i.e., depending on a user's role a well-defined set of customisation rules is used in the compilation. These well-defined customisation sets are the 'module types' referred to in Section III-A.1. The alternative to typing is to use individual rules from different sets, which might lead to 'feature-interaction'. Customisation types have unique identifiers, called customisation type identifiers (CTIDs). After compilation, *bygwyn* generates a signed compilation record containing both the CTID and the MD5 of the object code, explicitly binding the code to a type. Observe that the compilation rights are similar to the loading rights, but that the two policies are decoupled, so that, theoretically, users may generate code that they will not be able to load. Given this, we allow security policies to be specified of the form 'a user with authorisation X is allowed to load code that is compiled with customisation Y'. This is the basis of our code loading solution. Once loaded, the code runs natively at full speed, containing as many or as few runtime checks as necessary for this role.

Figure 2 shows a user who wants to push a home-grown module in the kernel. He offers the source together with his credentials to the compiler. The compiler selects, based on the credentials, the appropriate set of customisations to include in this compilation and compiles the code. If neither the extra rules, nor *bygwyn*'s internal rules generate any errors, an object file is generated and made tamper-proof by the compiler's compilation record.

2) 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. These routines are linked with the user code and reflect the role that is played by the kernel module. For example, students in a course on kernel programming may get

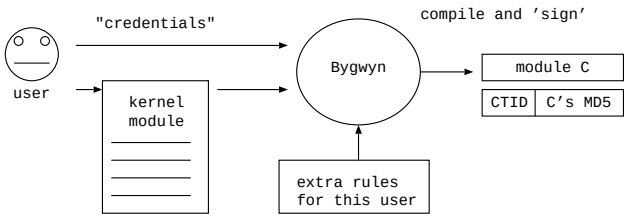

Fig. 2. User compiles kernel module

access to a function that allows them to read the kernel's `jiffies` counter, while network monitors get access to routines that allow them to examine an incoming or outgoing packet, e.g., regarding its headers. In other words, such routines are used to encapsulate the rest of the kernel. This is illustrated in Figure 3. In the figure, some function calls (`foo`) are really relegated to a wrapper, while others (`bar`) may be called directly. Access to kernel data structures is regulated similarly.

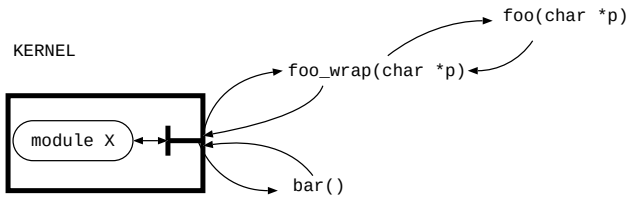

Fig. 3. The kernel is encapsulated behind an interface

D. Restricting Cyclone

Cyclone was created in an attempt to make C safe. However, some of the features of the language (e.g., exceptions, external linkage) still cannot be used safely within the kernel and have to be restricted. Exceptions have to be restrained so that they cannot 'leak' into the kernel. Also, there has to be a method of limiting the resources used by the code, in terms of instruction counts, stack space, heap space, and other kernel resources.

The method we use to solve these problems requires that the untrusted code is fully contained within a single translation unit. This way, we can globally analyse the code to decrease the amount of dynamic checks that need to be done. Before compilation, the translation unit has *environment setup code* (ESC) prepended to it, which is customly generated according to the credentials of the user requesting the compilation. The environment setup code defines the APIs that are available to the untrusted code, and then applies some language restrictions for the remaining code using language constructs that we created specially for this task. The code that is written after the restrictions are applied is effectively forced to be safe.

After the ESC has defined the available APIs, it removes the construct `extern "C"` from the language using the `forbid extern "C"` construct. It also opens a unique, randomly generated namespace for the untrusted code, to prevent any namespace clashes with other modules or with the kernel and to prevent any unauthorized imports of symbols from other namespaces. This leaves the untrusted code that follows fully isolated, solving all safety issues that are related to external linkage at no load-time cost.

The ESC declares kernel APIs and other functions and variables. Some of these are only for internal use by the setup code, so these declarations must not be made available to the untrusted code. Using the `forbid namespace` construct, the environment setup code can forbid access to certain namespaces, making it possible to create a private implementation section. The environment setup code makes everything unavailable to the untrusted code that it should not be trusted with, leaving the untrusted code with only the safe API wrappers that it is supposed to have access to.

At every entry point to the untrusted code, we automatically add some wrapper code to do resource cleanup after the code has been executed. This wrapper code also has to prevent the leakage of Cyclone exceptions into the kernel. The environment setup code can configure this code using the `wrap extern` construct. The wrapping code takes care of freeing any resources that were not released by the untrusted code, e.g., kernel locks. Our extended Cyclone compiler detects all functions that are potential entry points and automatically wraps these functions using the wrapper code declared by the ESC. Note that this wrapper mechanism is complementary to the wrappers applied by the ESC to the kernel APIs: this mechanism wraps the entry points from the kernel into the module, while the API wrappers wrap the entry points from the module into the kernel.

When we detect that the untrusted code misbehaves, we use the Cyclone exception mechanism to interrupt the code. The untrusted code is prevented from catching the thrown *misbehaviour exceptions* itself, because the ESC specifically forbids this using the language construct `forbid catch`. Misbehaviour exceptions are caught by the entry point wrapping code, which then makes sure that the misbehaving *OKE* module is removed from the kernel.

To make the ESC perform better, as well as to enable it to perform certain necessary but potentially unsafe actions, parts of it need to be written in C. In order to allow this, we added the `inline "C"` construct to the compiler. The C code declared using this construct is compiled with the module, giving the environment setup code all the freedom it needs.

In the kernel we have to restrict the stack usage of un-

trusted code, because stack overflows may lead to serious problems. To restrict the stack usage, we determine the stack frame size of every function. We then do a static call graph analysis to determine the maximum potential stack usage, and if it exceeds a certain limit, an error is generated at compile-time. When there is a potential recursion, the stack usage is of course potentially infinite. Therefore, when this happens, we do not generate a compile-time error but we insert a run-time check somewhere in the recursion loop to prevent exceeding the usage limit.

Finally, there have to be CPU cycle limitations to prevent infinite loops from hanging the kernel. We implemented this at almost no run-time cost by modifying the operating system's timer interrupt. When the timer interrupt has been fired a certain configurable amount of times after a segment of untrusted code has been entered, and the untrusted code has not finished yet, the return address of the interrupt is patched to a location that contains code that generates a Cyclone exception, so that when the interrupt returns the untrusted code is immediately stopped. Of course, the return address is *not* patched when the untrusted module is calling a kernel API function: in this case a flag is set, indicating to the kernel API wrapper function that it should immediately throw an exception when control returns from the kernel.

1) Sharing kernel memory using regions: Cyclone enforces a *region-based* memory protection system. In this system, every variable is bound to a logical region, which can be a program block (if the variable is declared there), the global region (for global variables) or the heap region (for dynamically allocated memory). Pointer variables are also in a region, and the values that pointer variables can point to are restricted by the region in which the pointer variable resides: pointers can only point to variables which reside in their own region, or which reside in a region which *outlives* their own region. A region X outlives a region Y when all variables in region Y are guaranteed to be destroyed before any variables in region X are destroyed. This pointing restriction makes sure that a pointer variable is always destroyed before the variable it points to is destroyed. For a more complete description of Cyclone's region system, see [GMJ+02].

Using the region system, the untrusted code is allowed to share data structures with the kernel. We extended the system to allow for interoperability with normal C kernel code. This enables us to ensure that whenever the kernel holds pointers to memory that is allocated by a Cyclone module, this memory cannot simply be freed by either the module's code or by the kernel (which would create dangling pointers in the kernel or the module, respectively).

We use two distinct memory regions, the Cyclone heap region `'H` and the kernel memory region `'kernel`, to distinguish between kernel-owned and Cyclone-owned memory regions. We allow `'H` pointers to point to memory in the kernel region, but not vice versa. This is correct, because the kernel heap outlives the Cyclone heap: *OKE* modules are always unloaded before the kernel is shut down. However, it is *not* correct when we don't do something to prevent the kernel from freeing the memory.

To make pointers to kernel memory safe, and to ensure that heap memory is freed correctly, we built a simple precise mark-and-sweep garbage collector that is started just before untrusted code becomes active. Because the code is inactive at the time of garbage collection, we only have to look at global variables (and static variables in functions), and because most modules in the kernel have a very limited number of global pointer variables, the time taken by garbage collection ranges from very little to almost none. Pointers in the modules that point to freed kernel memory are nullified at garbage collection time (if they are nullable – otherwise the module is unloaded). This is guaranteed to be safe because Cyclone always guards the dereferencing of nullable pointers with a run-time check. The garbage collector defers the freeing of kernel memory blocks until all modules that were active at the time of freeing have run a garbage collection round and therefore hold no more references to the memory block.

Deferring the freeing of kernel memory is only needed for modules that can potentially store pointers to kernel memory: many modules do not even have access to any APIs that allow them to get pointers to kernel memory with an actual `'kernel` region marker. For instance, the transcoding application we describe further on in this paper can only get its hands on pointers into an unspecified region. This prevents storage in a global variable, because Cyclone cannot deduce that the unspecified region outlives the global region: even though we know that the memory actually *is* in the `'kernel` region, the transcoder module does not need this information to perform its task.

The *type* of module defines the APIs the module has access to, and therefore it defines whether the module has access to actual pointers into `'kernel`. However, the *code* of a module controls whether garbage collection is actually needed. If the module does not use any global variables of pointer type, the module has no need for garbage collection. *Bygwyn* detects this situation and when it occurs, it disables garbage collection. The use of Cyclone *dynamic regions* provides support for the remaining needs for dynamically allocated memory. Dynamic regions are private memory heaps, bound to a dynamic program scope. The module can allocate memory in the

region only while it is inside the scope, and memory allocated in this way has a lifetime equal to the lifetime of the scope the region is bound to.

2) Access protection for structure members: Many kernel structures which can be shared between the untrusted code and the kernel partially contain sensitive data or data not intended for the untrusted module to modify or even read. Traditional ways to deal with this are to *anonymise* the fields, i.e., to empty them prior to giving the module access and restoring the values when the module cannot access them anymore, or to perform permission checks dynamically. However, both these solutions come with a performance penalty. Additionally, when the data structures are actually *shared* with the kernel, anonymising is not an option because the kernel still needs access to the data in question. For this reason, we decided to include support for *static* access permission checking in Cyclone, which is just as powerful and costs no performance. This access protection is implemented at the level of structure members. Regular Cyclone already allows making structure members `const` to make them read-only. We added the possibility of making structure members `locked`. Values of a `locked` type cannot be used in calculations, 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, locked types are limited to copying, and they cannot be read. This technique drastically reduces the need to anonymise data at run-time.

E. The legislator

Unlike the previous components, the *legislator* does not find itself at the core of the *OKE*. Rather, it is a tool that generates the customisations for the elastic language. The current *legislator* is extremely simple and should not be regarded as anything more than proof-of-concept. Even so, it enables one to customise the programming language in such a way that the remaining subset is 'safe' for kernel work [BS01]. In the *OKE* framework, *legislators* effectively define the module types, i.e., they are used to determine what the restrictions are that should be applied to the code. *Legislators* do *not* define the mapping between user roles and module types, i.e., they don't specify who gets access to which types. Here, as elsewhere, the *OKE* provides mechanisms, rather than policies.

IV. Applications

For illustration purposes, we now describe two areas in which the *OKE* has been applied: network monitoring and transcoding.

A. Network monitoring

Suppose we want to measure how many of our IP packets are fragmented. Instead of running a monitoring application fully in user space (e.g., on top of BPF in the kernel), it would be more efficient to inject such code straight into the kernel. The monitor takes the measurements it requires, correlates and aggregates the data ('x packets seen, of which y were fragments'), and only periodically reports the results back to a user-space application for further processing. Using the *OKE*, third parties are able to inject passive monitoring modules in kernel space, subject to resource restrictions and access constraints. Instead of determining *a priori* what sort of information may be monitored and offered to the user (as done by, for example, SNMP, and RMON), or even how many bytes are captured, we allow users with the appropriate credentials to measure anything they like. They may count and inspect entire packets, as long as they stick to the (parts of the) packets to which they have been given access.

1) Architecture: The open monitoring architecture (OMA) is sketched in Figure 4. The boxes labelled 'userspace application' and 'my measure module' are written by the users, all the other bits form part of the OMA. The memory mapped storage provides the client's user space application with a way to perform fast read and write operations on large log files in a sequential manner. It was adapted from a home-grown video server and provides users with a memory mapped window (of fixed size) which automatically slides over a log file when reading or writing.

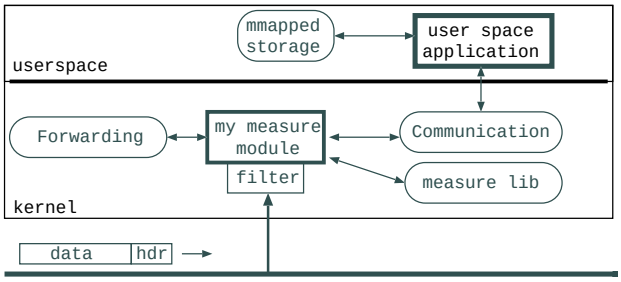

Fig. 4. Traffic monitoring in the kernel

A user's measurement code is compiled by *bygwyn* using customisation rules specific to the OMA, e.g., rules that automatically add filter code to the module. The role-specific filter determines which (bits of) packets the code may access. For example, for a specific role the filter may only supply the user with the first 28 bytes of all packets sent to a specific UDP/IP destination. By using `const locked` fields where needed when passing the packet to the modules, we are able to prevent unwarranted access to specific fields of the headers at no extra overhead. This allows us, for instance, to anonymise the source IP address

of a packet. Once loaded (see Section III-A), the module's filter registers an interest in network packets by means of a filter registration component on top of Linux Netfilter hooks. This means that when a packet is received, all registered filters are called in turn. A filter then determines whether the packet should go to its module and if so, calls the function `OKE_monitor_handle_packet` in the user's code to allow it to deal with it.

The user-installed code is supported by special-purpose kernel modules, e.g., a measurement library module for convenient access to packets, and a module implementing mechanisms for communicating with the user-space application. On the left we also see a 'forwarding' module. Using this module, we are able to tunnel a copy of the packet to a processing node nearby, instead of processing them locally. Time-critical functions (e.g., those in the measurement library) can be inlined with the client's code. Currently, the communication module is just a convenient wrapper around an `ioctl` channel.

2) Results: We have tested the OMA in filtering experiments, as reported in [BS01]. Comparing the per-packet processing time for four different filters of increasing complexity showed that the *OKE* performed only marginally worse than a filter implemented in pure C, and several times better than the same filters implemented in BPF (more details can be found in the paper). Filtering, however, is a fairly simple task, and the real question is whether we can come to good performance when the amount of processing that needs to be done on the packet increases? For this purpose, we implemented a transcoding architecture on top of the OMA and measured the performance of various types of transcoders. This transcoding architecture is discussed in the next section.

The most interesting observation, of course, is that *OKE* monitoring modules may process packets without having to copy them to user space. This begs the question of just how expensive a copy to user space is. To answer this, consider Figure 5. All measurements were taken on a 650 MHz Pentium-III running Linux (kernel version 2.4.0). The top line in the figure plots the per packet overhead in milliseconds from the kernel hook to an application in user space using the convenient, but somewhat inefficient `libipq` for various packet sizes. The `libipq` implementation provides a simple way to send packets caught in the Linux kernel for userspace via unicast netlink sockets. The times shown were measured from the time a packet was queued for userspace in the kernel (`NF_QUEUE`) until it was received by the application. It is interesting to note that the overhead increases but little with the size of the packets, suggesting that the bulk of the overhead is caused by factors other than the physical copy, probably context switching.

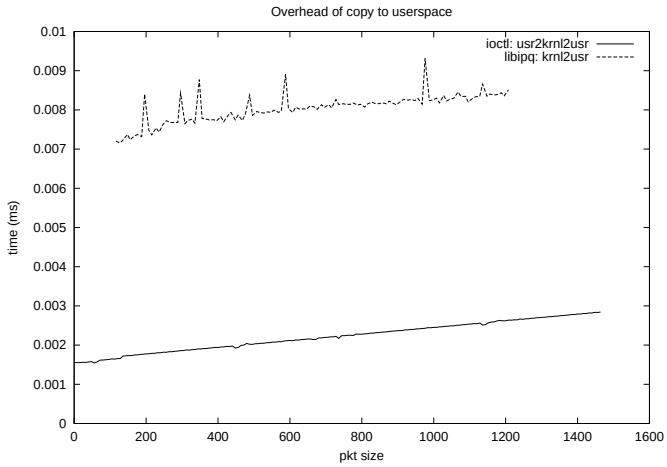

Fig. 5. Transferring data to userspace

`Libipq` is not the most efficient means of moving data from the kernel to user space. In order to get a more realistic lower bound on the time it takes to copy across the kernel boundary, the bottom line in the figure shows the time we measured for copying packets of various sizes to userspace over a dedicated `ioctl` channel. It should be mentioned that for this plot the initiative was on the side of the application, rather than the kernel, in that the application sends a copy request to the kernel module, which responds by copying the requested amount of data to the application's buffer. The plot shows the duration all the way from making the request to receiving the data. It is interesting to note that the overhead for requesting a copy of 0 bytes is approximately 1.5 μsecs per packet. Effectively this is the minimal overhead of crossing the kernel boundary twice (one round-trip time).

We emphasise that the two lines do not allow for fair comparison. For example, unlike the plot for the `libipq` implementation, the plot at the bottom does not show the time it takes to queue the packet (the kernel must temporarily store it in its own buffers), nor does it show how much time it takes between the queueing in the kernel and the reception in user space (e.g., scheduling the process). On the other hand, the `ioctl` plot includes the overhead of first sending a command from the application to the kernel, while the `libipq` implementation doesn't.

We would also like to stress that we have only shown the overhead for pushing the packets *up*, which often is only half of the trajectory. When packets need to be forwarded, they also must be pushed back into the kernel. This will add even more overhead (in the best case, using `ioctl`, the overhead will be exactly the same as for pushing the data up). Breaking down the packet transfer overhead indicated that almost all of it was caused by con-

text switching, but that even the physical copy itself took more than a micro-second.

The significance of the above analysis is that it provides us with a handle on when it is or isn't useful to do in-kernel processing. The conclusion is that whenever the overhead of crossing the kernel boundary is significant in comparison with the processing on the packet, it would be worthwhile to do such processing in the kernel, provided we are able to do so safely and efficiently.

B. Transcoding

Building on the monitoring platform discussed above, we implemented an open transcoding architecture (OTA). Essentially, the OTA uses the exact same filtering construction and components as found in the OMA, but unlike the monitoring platform, it removes the packets from the forwarding path. Packets may then be modified by the user's module and, after having been processed, reinserted in the forwarding path. In this way, users may transcode the packet data (and even the header), subject to the same sort of constraints as found in the OMA.

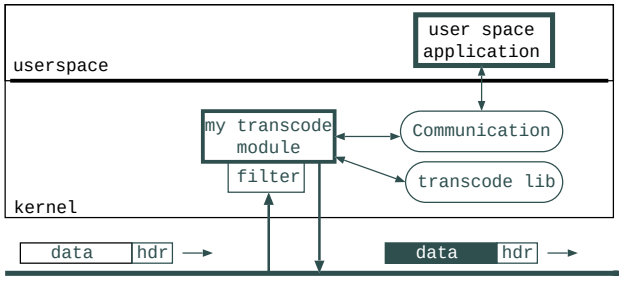

Fig. 6. Transcoding in the kernel

The transcoding library contains functions for convenient access to the packets, as well as functions that are commonly used by transcoders, e.g., the calculation of checksums, etc. Transcoding may result in packet growth, e.g., in the case of forward error correction. As long as the transcoding does not increase the packet length beyond that of the old packets length plus the available slack space, all transcoding can be done 'in-place' (zero-copy). Otherwise, a single copy is needed. If the packet grows beyond the MTU size, it is fragmented.

In our first transcoding example, we augmented an existing transmission with forward error correcting (FEC) code. To each UDP packet that is forwarded by an *OKE* router and which matches the filter, we add enough redundancy (by way of Hamming codes) to allow the receiving party to repair bit errors. As packets always grow and the available slack space is almost non-existent in our (standard) Linux kernel, this always results in a copy. From packet size of 800 onwards (approximately) we also suffer from packet fragmentation. Figure 7 compares the performance of the *OKE* transcoder as implemented in our modified Cyclone, with that of an optimal solution in C. It is interesting to note that there is no increase in the overhead when we start fragmenting the packets. This is due to (1) the fact that since packets always grow beyond the slack space, we always do a copy anyway, and (2) the fact that the overhead of this small copy is hardly significant compared to the processing overhead. In fact, it can be shown that fragmenting the packet is occasionally *good* for performance. The reason is that for certain packet sizes we are able to send the original packet payload without any modification, putting the redundant bits in the second fragment. Reinserting the original packet in the forwarding engine, while performing a small copy for the FEC, may outperform the longer copy which is required for sending the redundancy with the original payload within a single packet. In the latter case, we really have to copy the original payload to the new packet location.

The top two lines in Figure 7 show the overhead of FEC without loop unrolling, while the bottom two lines show how the same FEC processing benefits from aggressive optimisation due to manual loop unrolling. With the loop unrolling we were able to optimise away almost all of the runtime checks made by our modified version of Cyclone, so that its performance is just as good as the version implemented in C.[3] We think it is not unreasonable to assume this level of manual optimisation because the code running in the *OKE* operates under strict resource constraints, imposed by the *OKE* policy or by the nature of the application (e.g., the speed needed to handle packets at very high bandwidth). It is important to note in the plot that the overhead of the *OKE* implementation over the native C version is 10% in the worst case (at a packet size of 1470 bytes). In contrast, any implementation in user space would incur an overhead of almost 3 μs or 13% by copying 1470 bytes to user space (using the `ioctl` method), and a similar overhead copying the data back from user space. This shows that the *OKE* solution outperforms a userspace solution that uses the C implementation for the transcoder logic.

Figure 8 compares the performance of the *OKE* with that of optimal C code for a simple audio resampling transcoder. Here we resample audio packets to compress them in the face of increasing network loads to only half the number of bits. The significance of this example is

[3]We were actually a bit surprised by these performance figures and analysed the assembly generated by *gcc*. We found that due to what we suspect to be an anomaly in *gcc*'s optimisation, it mistakenly tried to optimise the C version in a way that actually *hurt* its performance a little. Because the *OKE*'s code is different this effect did not occur in the Cyclone implementation.

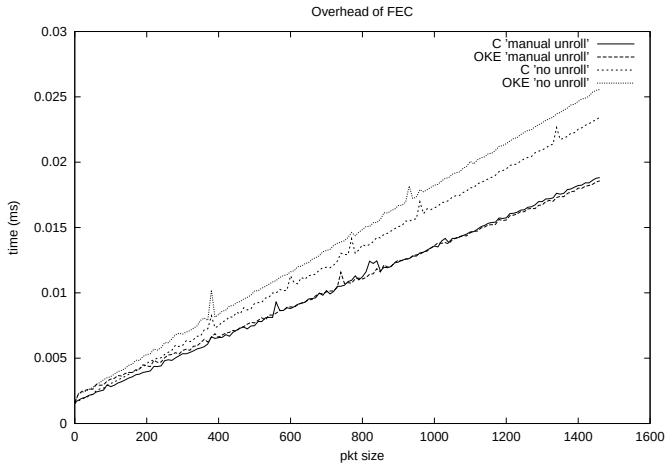

Fig. 7. FEC processing in C and in Cyclone

that in this case, the transcoder causes packets to shrink and hence allows for processing 'in-place'. It shows that the *OKE* implementation has an overhead over the C implementation of about 30% for small packets, and that this overhead diminishes as packet sizes grow. More importantly, in an equivalent userspace solution, the overhead of the copy to userspace (using the `ioctl` implementation) would be 1.5 μs or around 100% for small packets, and for packets of 1470 bytes still almost 3 μs or 37% (and probably higher, because this does not include the overhead of copying the data back into kernel space), which is much higher than the overhead of the *OKE* implementation.

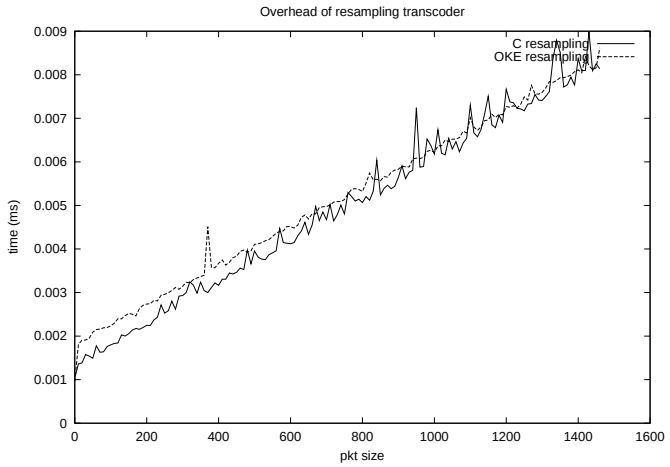

Fig. 8. Resampling audio packets in the kernel

1) Specific performance issues: Cyclone uses dynamic checks to verify some safety properties of the code, so Cyclone code must be slower than the equivalent C code. The Cyclone compiler mostly translates Cyclone code into similar C code with the necessary checks inserted, and we therefore expect the slowdown to be small.

Preliminary measurements we did on some test applications showed an overhead of sometimes up to 50%, which was higher than we expected. Analysis of the generated code showed that the culprit was the incrementing of pointer values using expressions like `++p`, which caused performance problems when used in inner loops. Such expressions were translated correctly by the Cyclone compiler, but the assembly code generated as a result by *gcc* 2.95 was terribly slow. The reason *gcc* was unable to optimise such expressions correctly was that in both C and Cyclone the *result* of an expression like `++p` can still be used, like in `*(++p)`, and in this case *gcc* did not optimise away the calculation of the result even though it was not used. The calculation of the result is sometimes nontrivial in this case because some Cyclone pointers include bounds information and are therefore tuples of values instead of single machine words. Especially in packet transcoding applications, pointer arithmetic is needed very frequently (as even array indexing involves pointer arithmetic), so we had to find a workaround for this problem. As a solution, we modified the Cyclone compiler to generate different code depending on whether the result of such a statement was actually used. This resulted in a drop of the overhead from 50% to less than 10%, which was more in line with our expectations.

2) Optimisations used: Additionally, we found that using some simple optimisation guidelines, the performance could be improved considerably. Cyclone provides so-called *non-nullable* pointers, which can neither be `NULL` nor out-of-bounds and which have a fixed upper bound that is known statically. Non-nullable pointers are declared using `@` instead of the usual `*` to distinguish them from nullable pointers. The use of non-nullable pointers never incurs a runtime check (in optimised code) as long as the array indexes used on them are constants. On the other hand, when arrays of unknown size are involved, Cyclone uses 'fat' pointers that contain bounds information as well as the pointer value, allowing for safe use of pointer arithmetic. These pointers are declared using `?` instead of `*` or `@` to distinguish them from other pointer types. Dereferencing these fat pointers always incurs a runtime check, although the optimiser can usually remove double checks. Cyclone provides these types together with the normal `*` pointer, which is similar to the non-nullable pointer but has the capability of being `NULL` as well. When working with pointers, we attempted to convert pointers into non-nullable pointers as early as possible, especially in inner loops. For example, instead of using `a[n] + a[n+1]` on a nullable pointer declared as `char ?a`, we would convert `a` into a non-nullable pointer of fixed length 2 using the declara-

tion `char @{2} a_n = a+n`. We would then replace `a[n] + a[n+1]` by `a_n[0] + a_n[1]`, which incurs *no* run-time checks compared to two checks for the original statement. The total number of checks is reduced from two to one, because the conversion into a non-nullable pointer *does* incur a run-time check.

V. Conclusions

We have described the *OKE* and its programming model. It was shown that trust management, language customisation and a trusted compiler allow users to add functionality to the kernel in a safe manner. In networking this is useful, e.g., to avoid unnecessary traversal of the kernel-userspace boundary. We briefly discussed example applications in networking and showed that the *OKE*'s performance is quite good. We also demonstrated that there exist very realistic network applications that benefit highly from execution in the kernel.

A. Discussion: active networks

We are currently considering the application of the *OKE* to the field of active networks. However, there are two main problems here, namely heterogeneity and scalability. The first problem is obvious: all the object code is very specific to our platform (Linux PC). It may be possible to ship source code instead of object code, or to provide multiple formats, but these solutions are not very elegant. The other problem is that in a large, distributed environment, it may be difficult to trust compilers running in foreign domains to generate code to run inside your kernel. Again, we may ship code in source format so we only need to trust our own compiler. Alternatively, we may agree on specific, trusted compilation sites (or repositories for module types) whose signatures are trusted by many (the VeriSign model). The current *bygwyn* already allows remote clients to access it via RPC, so that remote compilation is possible. For now, however, we see a use for the *OKE* mainly in localised environments, such as within or maybe between departments and organisations.

VI. Acknowledgements

We would like to thank Erik de Vink, Kostas Anagnostakis, Sotiris Ioannidis and Jonathan Smith for their useful comments and Jay Lepreau for helping us improve this paper. We also thank the anonymous reviewers.

References

[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.

[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.

[BHL00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in java. In *Proc. of 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[Bor94] N. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP International Conference*, Barcelona, Spain, 1994.

[BS01] Herbert Bos and Bart Samwel. The open kernel environment. In *OPENSIG'2001*, London, September 2001.

[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.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *OSDI*, 2000.

[FFF+97] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: a pedagogic programming environment for Scheme. In *Proc. 1997 Symposium on Programming Languages: Implementations, Logics, and Programs*, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. *Addison-Wesley ISBN 0-201-63451-1*, 1996.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, June 2002. To appear.

[HK99] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

[JMG+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.

[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.

[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.