Implementing Reliable Linux Device Drivers in ATS

Rui Shi
Boston University
shearer@cs.bu.edu

Abstract

Contemporary software systems often provide mechanisms for extending functionalities, which imposes great safety concerns on the well-being of critical infrastructures. ATS is a recently developed language with its type system rooted in *Applied Type System* framework (Xi 2004) which combines linear and dependent type theories for enforcing safe use of resources at low-level. In this paper, we describe a framework for constructing reliable Linux device drivers in ATS. Specifically, drivers are written and type checked in ATS, then compiled and linked to kernel with safety guarantee. Our preliminary experience shows that this approach can effectively enhance the reliability of device drivers and save the testing/debugging time.

Categories and Subject Descriptors D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; D.4.5 [*OPERATING SYSTEMS*]: Reliability

General Terms Languages

Keywords Device Driver Programming, Applied Type System, ATS

1. Introduction

Contemporary software systems such as operating systems and web servers often provide mechanisms for extending functionalities thus adapting the behavior of the systems. For instance, a device driver extends an operating system with capabilities of configuring and manipulating a particular hardware.

However, such extensions also introduce a large number of bugs as well as security holes into the system. For instance, in Windows XP, drivers account for 85% of recently reported failures. As shown by Chou et al. (2001), device drivers have error rate up to three to seven times higher than the rest of the kernel. Moreover, operating system extensions are often loaded into the kernel. Any fault in a kernel extension can corrupt the kernel data, causing the entire system to crash. Also, it is well-known that traditional software engineering approaches such as testing and debugging do not work very well on extensions due to concurrency and nondeterminism in the kernel.

In this paper, we focus on implementing more reliable device drivers using type-based techniques. The primary goal is to ensure the safe manipulation of various resources by extensions in the host system. While our system is designed for device drivers, we believe that our techniques can readily generalize to other kinds of extensions. More specifically, we propose to implement device drivers in ATS, and the type system of ATS is able to rule out a number of safety violations (e.g. *memory leak*, *buffer overflow*, *race conditions*, etc.) at compile time. Our approach is fundamentally different from previous systems that isolate drivers in a protection domain or extensively inserting run-time checks. In contrast, drivers written in ATS is proved to be safe statically and neither hardware support nor OS changes is needed. We design our system with the following criterion:

- **Fine-grained type-based resource manipulation:** We use a specification (proof) language to describe resources and enforce resource usage protocols at compile time. For sophisticated resources such as memory, our system is able to statically guarantee that no dangling pointers can ever be de-referenced, and addresses out of an array's bound will not be accessed at runtime. Moreover, linear types can statically prevent leak of resources (e.g. memory allocated by *kmalloc*), which is crucial for the safety of critical infrastructures.
- **Graceful error handling:** Kernel extensions deserve more safety concerns than user-level applications especially on error handling. For instance, if a module exits silently without properly releasing held resources upon some failure, the resources will be leaked permanently, which may cause kernel panic. In our system, driver developers are enforced to check the return code of each kernel API and handle possible errors according to different error code. This is precisely the style of *defensive programming*.
- **Safe concurrency:** Concurrency is ubiquitous in operating systems. For instance, different applications may access a hardware concurrently through a device driver which maintains some shared hardware specific data. Therefore, developing reliable extensions must take into account the potential safety violations caused by concurrency such as race conditions. Specifically, we use locks to protect shared data and the types assigned to locks can reflect the resource they are protecting.
- **Low run-time overhead:** Unlike implementing device drivers in type safe languages like Java, which relies on garbage collection, our system will translate programs written in ATS to C source programs and use native C compiler to compile the obtained C code. Therefore, we expect the performance be comparable to drivers directly written in C. On the other hand, as we take a pure static approach, the drivers implemented in our system are expected to outperform the dynamic approaches that rely on run-time isolation of protection domains (Swift et al. 2003), where kernel/extensions boundary crossings are costly.

The rest of the paper is organized as follows. In Section 2 we give an overview of the main techniques of our approach. We then present a case study on implementing a Linux device driver

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLPV'07, October 5, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-677-6/07/0010. . . $5.00

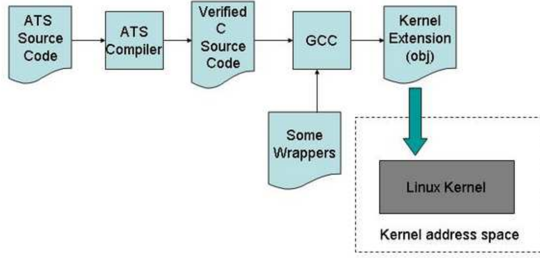

Figure 1. Architecture: Implementing Linux device drivers in ATS

in Section 3. Lastly, we discuss some closely related work then conclude. Due to the space constraint, it is impractical to present theoretical justifications of our approach. Rather, we will rely on informal but intuitive explanations. No prior familiarity of ATS is required.

2. Our Approach

Platform We build our own kernel using Linux source distribution 2.6.15 and implement the system upon this kernel. All experiments are conducted using this kernel.

Linux Device Driver Model Device drivers play a special role in the Linux kernel. They make a particular piece of hardware respond to a well-defined internal programming interface. User activities are performed by means of a set of standardized calls (e.g. `fread`) that are independent of the specific driver. Mapping those calls to device-specific operations that act on real hardware is the role of the device driver. Drivers can be built separately from the rest of the kernel and plugged in at run-time when needed. For instance, the following C code is to configure a network interface device driver named `snull` with specific operations:

```
dev->open  =  snull_open;
dev->stop  =  snull_release; ...
```

where `dev` is a pointer held by the kernel which points to the device specific data. The right hand side is a set of functions that are implemented by the driver writer. By assigning each field of `dev` a specific function, the kernel is thus able to configure the way to manipulate the `snull` device. For instance, the request to open the device is handled by the function pointed by `dev->open`, thus, the function `snull_open` is executed.

Architecture The architecture of our system is presented in Figure 1. The basic steps are sketched as follows:

1. Define the basic data structures of the device driver in C. These definitions usually include some device specific parameters and constants.
2. Implement driver functionalities in ATS. In particular, views (a form of specification) (Zhu and Xi 2005) and types are used to describe resources and specify safety policies through pre/post conditions of functions so that the type system is able to check that the implementation does respect the safety property.
3. Type-check and compile all ATS source programs to corresponding C source programs using the ATS compiler. The obtained C source programs are guaranteed free of type errors and resource misuses.
4. Compile all C programs (existing + obtained from ATS) to yield an object module using GCC.
5. Link the compiled module to the kernel. Linux provides mechanisms to load modules to the kernel at run-time. For instance, *insmod* is used to load a module into the kernel while *rmmod* is used to remove a module from the kernel.

Kernel API Wrappers Note that our system mainly focuses on the reliability of extensions (i.e. drivers) and it is apparently impractical to verify the entire kernel in ATS. We thus choose to trust the existing kernel and assume the correctness of kernel APIs. We believe this assumption is reasonable because the majority of kernel failures are due to extensions instead of the kernel itself. In practice, kernel APIs can be ascribed ATS types that more precisely capture the invariants of C functions. For instance, in Linux, *copy_from_user* is a function used to move data across the protection domain from the user space to the kernel space. In order for *copy_from_user* to be used safely, programmer must follow certain safety requirements. However, C type system can perform little static or dynamic checking on the use of *copy_from_user*. In ATS, we can build a wrapper for this function and assign it the following type[1]:

$$\forall l_k : addr.\forall l_u : addr.\forall m : nat.\forall n : nat.\forall c : nat(n \geq c) \supset \\ (!\mathbf{bytes}(l_k, n), !\mathbf{bytes}(l_u, m) \mid \mathbf{ptr}(l_k), \mathbf{ptr}(l_u), \mathbf{int}(c)) \\ \rightarrow \exists c' : nat.(c' \leq c) \wedge \mathbf{int}(c')$$

There are a number of safety properties enforced by the above type. Two views $\mathbf{bytes}(l_k, n)$ and $\mathbf{bytes}(l_u, m)$ serve as the preconditions, which denote that there are two byte arrays residing at the addresses[2] l_k and l_u with size n and m, respectively. The syntax ! used in the above type means that the view is preserved across the function call (otherwise view changes need to be specified as post conditions). Note that the third argument (of type $\mathbf{int}(c)$) is the expected number of bytes to transfer and the guard $(n \geq c)$ clearly states that the size of the destination buffer must be greater than the specified number. The function returns the actual number of bytes transferred which is of type $\mathbf{int}(c')$ for some c', where the assertion $(c' \leq c)$ states that the returned number is less than the expected one. The counterpart of *copy_from_user*, *copy_to_user*, which transfers data from the kernel space to the user space can be assigned a similar type in ATS.

Note that assigning a foreign function a type in a type safe language is usually referred as *Foreign Function Interface* (FFI). For instance, both Java and Ocaml support FFI for effectively interfacing with an unsafe language (efficiency) and reusing legacy code (extensibility). However, in Java and Ocaml, the types assigned to foreign functions are usually not expressive enough to capture low-level invariants as in ATS (e.g. each argument of an FFI is assigned the type `value_t` in Ocaml). In contrast, ATS programs can directly interface with foreign functions such as *copy_from_user* and take advantage of the type checker to ensure the safe use of foreign functions. To our best knowledge, this design has never been explored or applied to extensions of mainstream OS before.

3. A Case Study of Scull Device Driver

We have prototyped a Linux device driver called *scull* (simple character utility for loading localities) in the current implementation of ATS. *scull* is a running example used as a template for writing real drivers for real devices in the book by Rubini et al. (2005).

[1] The syntax | is used for separating views from types.

[2] It is also possible for our system to statically differentiate the kernel space pointers from the user space pointers. We omit this for simplicity.

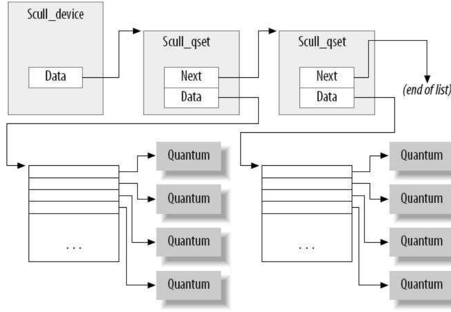

Figure 2. The memory layout of a scull device

```
struct scull_qset {
    void **data;            /* pointer to a quantum set */
    struct scull_qset *next; /* pointer to the next node */
}

struct scull_dev {
    struct scull_qset *data; /* pointer to qset list */
    int quantum;            /* current quantum size */
    int qset;               /* the current qset size */
    unsigned long size;     /* amount of data stored */
    struct semaphore sem;   /* semaphore */
};
```

Figure 3. Data structures of scull device defined in C

In this section, we are to conduct a case study on how to implement this device driver in ATS, demonstrating some interesting aspects.

Device Layout In nature, *scull* is a character driver that acts on a memory area as though it were a device. Hence, as far as *scull* is concerned, the word *device* can be used interchangeably with *the memory area used by scull*. We visualize the layout of a *scull* device in Figure 2. Each *scull* device maintains a linked list, each node of which points to an array of `qset` intermediate pointers. And each intermediate pointer points to an area of `quantum` bytes. Note that `qset` and `quantum` are two positive integers stored in the device control structure. In the following, we refer arrays holding raw data as *quantum* (the gray shadowed boxes in Figure 2) and those containing pointers to *quantum* as *quantum sets* (the white boxes separated by lines in Figure 2 and we use *qset* for short), respectively.

We present two C structure definitions of the *scull* device in Figure 3. The *scull_qset* structure simply defines a singly-linked list and the $scull_dev$ structure maintains the device specific parameters such as the size of each *quantum*, the size of each *qset* as well as a pointer to the head of a *scull_qset* list. Apparently, C provides little safety guarantee on manipulating the *scull* device due to its inaccurate (type) specification. For instance, the field `data` is assigned the type **void**** which can not reflect the fact that `data` should point to an array of length `qset` whose value is stored in $scull_dev$. In the following, we will show step by step how a variety of program invariants can be accurately specified by our specification language and ensured by the type checker statically.

To characterize the memory layout as well as capture program invariants, we immediately encounter the needs for representing the concept of bytes by defining a subset sort as follows:

```
sortdef byte = {a: int | 0 <= a, a < 256}
```

which basically states that bytes are integers between 0 and 255. We can then use the following concrete syntax:

```
dataview bytes (int, addr) =
  | {l: addr} bytes_none (0, l)
  | {n: nat, l: addr, b: byte | l > null}
      bytes_some (n + 1, l) of
        (byte(b) @ l, bytes (n, l + 1))
```

to declare a recursive view constructor **bytes**. Given an integer n of sort int and an address l of sort $addr$, the view **bytes**(n, l) states that there are n bytes consecutively stored in memory starting from the memory address l.

The attempt to describe a *quantum* using a bytes view simply fails due to the observation that each pointer in a *qset* does not necessarily point to a *quantum* that is dynamically allocated. We then adopt a notion called *optional view* to tackle this problem. For instance, the following dataview is introduced to reflect the dynamic nature of a *quantum*:

```
dataview bytes_opt_v (int, addr) =
  | {q: nat} bytes_opt_v_none (q, null)
  | {l: addr, q: nat | l > null}
      bytes_opt_v_some (q, l) of bytes (q, l)
```

Given a natural number q and a memory address l, the view **bytes_opt_v**(q, l) denotes that either there is no memory allocated if l is $null$ or there is an array of q bytes located at the address l if l is not null. The purpose of an optional view is to force programmers to perform necessary run-time checks (a style of *defensive programming*).

With the optional view, we can associate with the *qset* structure a view as follows:

```
dataview qset_v (int, addr, int) =
  | {l: addr, quantum: nat} qset_v_none (0, l, quantum)
  | {n: nat, self: addr, data: addr, quantum: nat
      | self > null, bptr >= null}
      qset_v_some (n + 1, self, quantum) of
        (ptr(data) @ self, bytes_opt_v (quantum, data),
         qset_v (n, self + word_size, quantum))
```

where *word_size* is a configurable static integer which denotes the the size of a word on the target machine. For instance, we set *word_size* to 4 in order to accommodate low-level reasoning on 32-bit platforms. As a *qset* is dynamically allocated as well, we can introduce an optional view **qset_opt_v** which is defined in a similar manner as **bytes_opt_v**.

In order to characterize the memory layout of scull device structures defined in Figure 3, we present some view definitions in Figure 4 and give brief explanations for each of them:

```
stadef word_size: int = 4

dataview qset_seg_v (int, addr, addr, int, int) =
  | {l: addr, qs: nat, qt: nat}
      qset_seg_v_null (0, l, l, qs, qt)
  | {n: nat, fst: addr, next: addr, lst: addr,
      data: addr, qs: nat, qt: nat | fst > null}
       qset_seg_v_some (n + 1, fst, lst, qs, qt) of
         (ptr(data) @ fst, ptr(next) @ (fst + word_size),
          qset_opt_v (data, qs, qt),
          qset_seg_v (n, next, lst, qs, qt))

viewdef qset_list_v (n: int, l: addr, qs: int, qt: int)
  = qset_seg_v (n, l, null, qs, qt)

viewdef scull_dev_v (self: addr, n: int, qs: int, qt: int)
  = '([data: addr] '(ptr(data) @ self,
                    qset_v_list(n, data, qs, qt)),
      int(qt) @ (self + word_size),
      int(qs) @ (self + 2 * word_size),
      [size: nat] int(size) @ (self + 3 * word_size))
```

Figure 4. View definitions of scull data structures in ATS

$$\begin{array}{lll} \textit{scull_trim} & : & \forall n : nat.\forall l : addr.\forall qs : nat.\forall qt : nat. \\ & & (l > null) \supset (\textbf{scull_dev_v}(n, l, qs, qt) \mid \textbf{ptr}(l)) \rightarrow (\exists qs' : nat.\exists qt' : nat.\textbf{scull_dev_v}(0, l, qs', qt') \mid \textbf{Int}) \\ \textit{scull_read} & : & \forall dev : addr.\forall n : nat.\forall buf : addr.\forall m : nat.\forall pos : addr.\forall ct : nat. \\ & & (ct \leq n) \supset (!\textbf{bytes}(n, buf), \textbf{int}(m)@pos \mid \textbf{ptr}(dev), \textbf{semaphore}(\textbf{scull_dev_v}0(dev)), \textbf{ptr}(buf), \textbf{int}(ct), \textbf{ptr}(pos)) \\ & & \quad \rightarrow \exists c : int.(\textbf{count_v}(c, m, pos) \mid \textbf{int}(c)) \end{array}$$

Figure 5. Types assigned to functions manipulating the *scull* device in ATS

- The qset segment view, **qset_seg_v**(n, fst, lst, qs, qt), models linked *qset* segments, where n is the number of segments in the list, fst is the starting pointer, lst is the ending pointer, qs and qt are sizes of inclusive *qsets* and *quantum*, respectively. Also, *qset_seg_v_null* models an empty segment as indicated by the length 0 while *qset_set_v_some* is used to model non-empty segments. The latter one states that there is a pointer stored at the address fst which may point to a *qset* and another pointer (of type **ptr**$(next)$) is stored at the address fst + *word_size* which points to the rest of the segments (of view **qset_seg_v**$(n, next, lst, qs, qt)$).
- The view **qset_list_v**(n, fst, qs, qt) is a special instance of qset segment view, which is defined as

  $$\textbf{qset_seg_v}(n, fst, null, qs, qt)$$

  Namely, the ending pointer is always $null$ for a qset list.
- The definition of **scull_dev_v** faithfully describes the memory layout of the corresponding C structure. It should be straightforward to relate the view definition with the C declarations in Figure 3. Note that the only difference is the treatment of the semaphore. In C, a semaphore is used for acquiring exclusive access of the scull device. However, there is no counterpart in the view definitions. Note that C has the flexibility to define the semaphore as a part of the scull device then uses just one pointer to access the semaphore as well as the device. Ideally, a semaphore should be used to guard the access to the scull device and the type system can guarantee that the semaphore has to be acquired before any access to the device. We therefore implement semaphore separately. A value of type **semaphore**(V) for some view V is a semaphore protecting the view V. We omit the formal study of semaphore due to the space constraint.

We now give two examples to show how views defined so far can be used to enforce safety properties for the *scull* driver. The actual code is omitted for brevity.

Exampe: Device initialization *scull_trim* is an initialization function, which, given a scull device, walks through the *qset list*, safely frees all the allocated memory in it and resets parameters. In practice, *scull_trim* is called when an application opens or closes the scull device. The type assigned to *scull_trim* is shown in Figure 5. The pre-condition **scull_dev_v**(n, l, qs, qt) requires that there is a $scull_dev$ structure stored at the address l which points to a qset list of length n with parameters set to qs and qt, respectively. The post-condition **scull_dev_v**$(0, l, qs', qt')$ asserts that a new *scull_dev* structure is stored at the same address l which points to a *qset* list of length 0, thus, an empty list. Note that the parameters are also reset after initialization.

Example: Reading from the scull device One important operation of character device drivers is *read* operation, which is called when an application call `fread` to read the *scull* device. We are to demonstrate how the *read* operation can be implemented in our system with high safety guarantee. We first define the following shorthand:

```
viewdef scull_dev_v0 (dev: addr) =
  [n: nat, qs: nat, qt: nat] scull_dev_v (dev, n, qs, qt)
```

The type assigned to *scull_read* is given in Figure 5. We provide some explanations as follows:

- The view !**bytes**(n, buf) states that a byte array with length n is located at the starting address buf. Also, this view is preserved across the function call as indicated by !. ct is the expected number of bytes to read. The guard $ct \geq n$ specifies that ct must not exceed the length of the buffer. pos represents the address where the current file position (maintained by the Linux kernel) is stored. So the view **int**$(m)@pos$ indicates that there is some natural number of type **int**(m) stored at the address pos.
- A pointer of type **ptr**(dev) points to the scull device and a semaphore of type **semaphore**(**scull_dev_v**$0(dev)$) is used to protect the access to the scull device. The view for the scull device can only be extracted by acquiring the semaphore first.
- **count_v** is a dataview constructor defined as follows:

  ```
  dataview count_v (int, int, addr) =
      | {m: nat, c: nat, l: addr | l > null, c >= 0}
          count_v_norm (c, m, l) of (int(m+c) @ l)
      | {m: nat, l: addr, c: int | l > null, c < 0}
          count_v_err (c, m, l) of (int(m) @ l)
  ```

  Intuitively, **count_v** is used to reflect very interesting invariants of the return value of *scull_read*. Namely, if no error occurs, the return value is the number actually read from the device. In addition, the value stored at pos should be increased by the number of exact bytes read. Otherwise, if some error occurs, the return value is negative and the file position should be kept unchanged. Note that the similar techniques are used frequently for properly handling errors in system programming.

4. Related Work and Conclusion

We have so far seen many successful applications of functional programming to compiler construction. However, it is still rare to find convincing uses of functional programming in building operating systems. In a recent study, it is demonstrated that Haskell (Peyton Jones et al. 1999) can be used to implement an experimental OS called House (Hallgren et al. 2005). There are also various attempts to apply language-based techniques for enhancing the reliability of the operating systems. For instance, Singularity (Fähndrich et al. 2006) is a recently developed experimental OS which, with the primary goal of dependability, heavily employs many research advances such as software isolation and contract-based message passing.

In contrast, with dependent types and linear types as well as support for theorem proving, ATS offers an effective approach to reasoning about resource usage in practical programming. In this paper, we describe some ongoing work on applying ATS to error-prone device driver programming. The key insight is that ATS can effectively describe resources as well as enforce many program invariants.

One impediment is the type (proof) annotations burden imposed on programmers, which seems inevitable if formal reasoning of nontrivial safety properties is desired. Recently, we have been investigating techniques for automatically inferring a large portion of proofs to reduce the burden. We hope these techniques could

greatly facilitate the construction of more reliable OS components as well in the future.

References

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 177–190, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-322-0.

Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. *SIGPLAN Not.*, 40(9):116–128, 2005. ISSN 0362-1340.

Simon Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available at `http://www.haskell.org/onlinereport/`, February 1999.

Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. Oreilly and Associates Inc, 2005. ISBN 978-0596-00590-0.

Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222, New York, NY, USA, 2003. ACM Press.

Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, Long Beach, CA, January 2005. Springer-Verlag LNCS, 3350.