To appear: OSDI 2006
Seattle, WA, November 6-8

SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques

Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak
University of California, Berkeley
{zf,jcondit,zra,ibagrak}@cs.berkeley.edu

Rob Ennals
Intel Research Berkeley
robert.ennals@intel.com

Matthew Harren, George Necula, Eric Brewer
University of California, Berkeley
{matth,necula,brewer}@cs.berkeley.edu

We present SafeDrive, a system for detecting and recovering from type safety violations in software extensions. SafeDrive has low overhead and requires minimal changes to existing source code. To achieve this result, SafeDrive uses a novel type system that provides fine-grained isolation for existing extensions written in C. In addition, SafeDrive tracks invariants using simple wrappers for the host system API and restores them when recovering from a violation. This approach achieves fine-grained memory error detection and recovery with few code changes and at a significantly lower performance cost than existing solutions based on hardware-enforced domains, such as Nooks [33], L4 [21], and Xen [13], or software-enforced domains, such as SFI [35]. The principles used in SafeDrive can be applied to any large system with loadable, error-prone extension modules.

In this paper we describe our experience using SafeDrive for protection and recovery of a variety of Linux device drivers. In order to apply SafeDrive to these device drivers, we had to change less than 4% of the source code. SafeDrive recovered from all 44 crashes due to injected faults in a network card driver. In experiments with 6 different drivers, we observed increases in kernel CPU utilization of 4–23% with no noticeable degradation in end-to-end performance.

1 Introduction

Large systems such as operating systems and web servers often provide an extensibility mechanism that allows the behavior of the system to be customized for a particular usage scenario. For example, device drivers adapt the behavior of an operating system to a particular hardware configuration, and web server modules adapt the behavior of the web server to the content or performance needs of a particular web site. However, such extensions are often responsible for a disproportionately large number of bugs in the system [9, 33], and bugs in an extension can often cause the entire system to fail. Our goal is to improve the reliability of extensible systems without requiring significant changes to the core of the system. To do so, we must *isolate* existing extensions, preferably with little modification, *restore system invariants* when they fail, *restart* them automatically for availability, and (ideally) *restore active sessions*.

In this paper, we focus on the specific problem of improving device driver reliability. Previous systems have attempted to address this problem using some form of lightweight protection domain for extensions. For example, the Nooks project [32, 33] runs Linux device drivers in an isolated portion of the kernel address space, modifying kernel API calls to move data into and out of the extension. This approach prevents drivers from overwriting kernel memory at the cost of relatively expensive driver/kernel boundary crossings.

Our system, SafeDrive, takes a different approach to improving extension reliability. Instead of using hardware to enforce isolation, SafeDrive uses language-based techniques similar to those used in type-safe languages such as Java. Specifically, SafeDrive adds type-based checking and restart capabilities to *existing* device drivers written in C without hardware support or major OS changes (i.e., without adding a new protection domain mechanism). We have four primary goals for SafeDrive:

- **Fine-grained type-based isolation**: We detect memory and type errors on a per-pointer basis, whereas previous work has only attempted to provide per-extension memory safety. SafeDrive ensures that data of the correct type is used in kernel API calls and in shared data structures. This advantage is critical, because it means that SafeDrive can catch memory and type violations be-

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0509544.

fore they corrupt data, even for violations that occur entirely within the driver. Thus, we can prevent the kernel or devices from receiving incorrect data for these cases. SafeDrive can also catch more memory-related bugs than hardware-based approaches; specifically, SafeDrive can catch errors that violate type safety but do not trigger VM faults. In addition, because errors are caught as they occur, SafeDrive can provide fine-grained error reports for debugging.

- **Lower overhead for isolation**: SafeDrive exhibits lower overhead in general, particularly for extensions with many crossings (for which Nooks admits it is a poor fit [33, Sec. 6.4]). Compared with SafeDrive, hardware-enforced isolation incurs additional overhead due to domain changes, page table updates, and data copying. Also, the stronger type invariants that SafeDrive maintains makes it possible to check many pointer operations statically.
- **Non-intrusive evolutionary design.** SafeDrive provides type safety without changing the structure of the host system (e.g., the OS kernel) significantly and without rewriting extensions to use a new language or API.
- **Protection against buggy (but not malicious) extensions.** In rare cases where true type safety would require significant changes to the extension or to the host API, we prefer to trust individual operations whose safety we cannot verify and gradually migrate to more complete isolation over time. For example, our current implementation does not yet attempt to verify memory allocation, deallocation, and mapping operations. In addition, we make no attempt to protect the system from extensions that abuse CPU, memory, or other resources (unlike OKE [5] and Singularity [27]). As a consequence, SafeDrive is able to guard against mistakes made by the author of the extension but does not attempt to protect against a malicious adversary capable of exploiting specific behaviors of our system.

C and its variants have a number of important constructs that can be used to cause violations. In addition to the most obvious issue of out-of-bounds array accesses, C also has fundamental problems due to unions, null-terminated strings, and other constructs. To transform a driver written in C (which includes all Linux drivers) into one that obeys stricter type safety requirements, we must fix all of these flaws without requiring extensive rewrites and ideally without requiring modifications to the kernel.

The existing approaches to type safety for C involve the use of "fat" pointers, which contain both the pointer and its bounds information. CCured [24], for example, can make a legacy C program memory-safe by converting most of its pointers into fat pointers and then inserting run-time checks to enforce bounds constraints. However, this approach is not realistic for drivers or for the kernel, since it modifies the layout of every structure containing pointers as well as every kernel API function that uses pointers. To use CCured effectively in this context, we would need to "cure" the entire kernel and all of its drivers together, which is impractical.

Instead, SafeDrive uses a novel type system for pointers, called Deputy, that can enforce memory safety for most programs without resorting to the use of fat pointers and thus without requiring changes to the layout of data or the driver API. The key insight is that most of the required pointer bounds information is already present in the driver code or in the API—just not in a form that the compiler currently understands. Deputy uses type annotations, particularly in header files for APIs and shared structures, to identify this information in places where it already exists.

Although adding annotations to kernel headers or driver code may seem like a burden, there are many reasons why this approach is a practical one. First, the required annotations are typically very simple, allowing programmers to easily express known relationships between variables and fields (e.g., "p points to an array of length n"). Second, the cost of annotating kernel headers is a one-time cost; once the headers are annotated, the marginal cost of annotating additional drivers is much smaller. Third, annotations are only mandatory for the driver-kernel interface, while the unannotated data structures internal to the driver can use fat pointers. About 600 Deputy annotations were added to kernel headers for the 6 drivers we tested (Section 5.2).

Any solution based on run-time enforcement of isolation must provide a mechanism for dealing with violations. In SafeDrive we assume that extensions are restartable provided that certain system invariants are restored. In the case of Linux device drivers, invariant restoration consists of releasing a number of resources allocated by the driver and unregistering any name space entries registered by the driver (e.g., new device entries or file system entries), both of which we will refer to as *updates*. In order to undo updates during fault recovery, we track them using wrappers for the relevant API calls. Because SafeDrive allows an extension to operate safely in the kernel address space without the use of a hardware-enforced protection domain, the task of managing and recovering kernel resources is greatly simplified.

As with Nooks, SafeDrive cannot prevent every extension-related crash, nor can it guarantee that malicious code cannot abuse the machine or the device. However, because SafeDrive can catch errors that corrupt the driver itself without corrupting other parts of the system, we expect it to catch more errors, and we expect it to

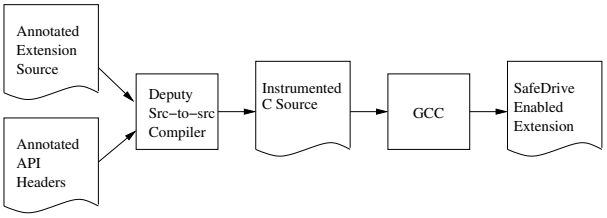

Figure 1: Compilation of an extension in SafeDrive.

catch them earlier.

We have implemented SafeDrive for Linux device drivers, and we have used the system on network drivers, sound drivers, and video drivers, among others. Our experiments indicate that SafeDrive provides safety and recovery with significantly less run-time overhead than Nooks, especially when many calls to/from the driver are made (e.g., 12% vs. 111% overhead in one benchmark, and 4% vs. 46% in another). The main conclusion to draw from these experiments is that language-based techniques can provide fine-grained error detection at significantly lower cost by eliminating the need for expensive kernel/extension boundary crossings.

In Section 2, we present an overview of the SafeDrive system, including the compile-time and run-time components. Then, in Section 3 and Section 4, we describe in detail the required annotations and the implementation of the recovery system. Section 5 describes our experiments. Finally, Section 6 and Section 7 discuss related work and our conclusions.

2 SafeDrive Overview

In SafeDrive, isolated recoverable extensions require support from the programmer, the compiler, and the runtime system. The programmer is responsible for inserting type annotations that describe pointer bounds, and the compiler uses these annotations to insert appropriate run-time checks. The runtime system contains the implementation of the recovery subsystem.

The compiler is implemented as a source-to-source transformation. Given the annotated C code, the Deputy source-to-source compiler produces an instrumented version of this source code that contains the appropriate checks. This instrumented code is then compiled with GCC. Most of the annotations required for this process are found in the system header files, where they provide bounds information for API calls in addition to the recovery system interface. The compilation process is illustrated in Figure 1.

At run time, a SafeDrive-enabled extension is loaded into the same address space as the host system and is linked to both the host system and the SafeDrive runtime system. Because all code runs in the same address space,

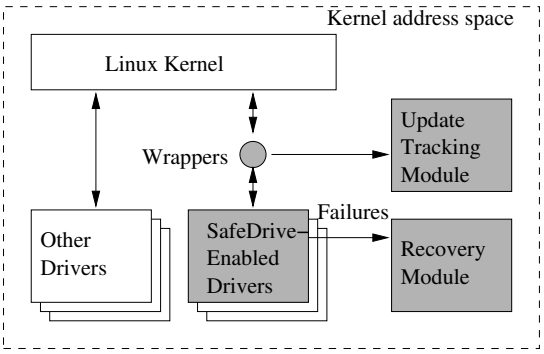

Figure 2: Block diagram of SafeDrive for Linux device drivers. Gray boxes indicate new or changed components.

no special handling of calls between the host system and the extension is required, apart from the run-time checks that the Deputy compiler inserts based on each function's annotations. The extension can read and write shared data structures directly, again using the Deputy-inserted run-time checks to verify the safety of these operations. Note that the host system does not need to be annotated or compiled with the Deputy compiler. Another consequence of this binary compatibility property is that non-SafeDrive extensions can be used with a SafeDrive-enabled host system as-is.

The SafeDrive runtime system is responsible for "update tracking" and recovery. It maintains data structures tracking the list of updates the extension has done to kernel state. Whenever a Deputy check fails, the control is passed to the SafeDrive runtime system, which attempts to cancel out these updates and restart the failed extension. Figure 2 shows the structure of the SafeDrive runtime system for the Linux kernel.

Although this paper focuses on Linux device drivers, we believe that the principles used in SafeDrive could be applied to a wide range of other extensible systems without a large amount of additional effort.

2.1 Example

Figure 3 shows some sample code adapted from a Linux device driver. Here the programmer has added the Deputy annotation "`count(info_count)`" to the `buffer_info` field, which indicates that `info_count` stores the number of elements in this array. The original code contained the same information in comments only; thus, the existence of this relationship between structure fields was previously hidden to the compiler.

Code in boldface shows run-time checks that SafeDrive inserts into the instrumented version of the file (see Figure 1) to enforce isolation and to support

```
struct e1000_tx_ring {
  ...
  unsigned int info_count;
  struct e1000_buffer * count(info_count)
      buffer_info;
  ...
};

static boolean_t
e1000_clean_tx_irq(
    struct e1000_adapter *adapter,
    struct e1000_tx_ring *tx_ring)
{
  ...
  assert(tx_ring != NULL);
  spin_lock(&tx_ring->tx_lock);
  track_spin_lock(&tx_ring->tx_lock);
  ...
  i = tx_ring->next_to_clean;
  assert(0 <= i && i < tx_ring->info_count);
  eop = tx_ring->buffer_info[i]
                   .next_to_watch;
  ...
  track_spin_unlock(&tx_ring->tx_lock);
  spin_unlock(&tx_ring->tx_lock);
}
```

Figure 3: Example adapted from the Linux `e1000` network card driver. The one programmer-inserted annotation is underlined. The boldface code shows Deputy run-time checks that are inserted into the instrumented version of the file.

recovery. The first and third checks are assertions that enforce memory safety. The first check ensures that `tx_ring` is non-null, which is required because Deputy assumes that unannotated function arguments are either null or point to a single element of the declared type, much like references in a type-safe language. The third check in the example enforces the bounds declared for the `buffer_info` field before it is accessed. Note that a simple optimization ensures that we do not insert redundant checks every time `tx_ring` is dereferenced; indeed, the low overhead of SafeDrive is in part due to the fact that most pointer accesses require very simple checks or no checks at all.

If either of these assertions fail, SafeDrive invokes the recovery subsystem. To support recovery, SafeDrive inserts code to track the invariants that must be restored, as seen in the second and fourth boldface statements in this example.

This example shows that SafeDrive preserves the binary interface with the rest of the kernel, inserts relatively few checks (tracked resource allocation is rare, and most pointers point to a single object), at the expense of a few annotations that capture simple invariants. In the next two sections, we describe in detail our type system and the associated recovery system.

3 The Deputy Type System

The goal of the Deputy type system is to prevent pointer bounds errors through a combination of compile-time and run-time checking. Adding these checks is a challenging task because the C language provides no simple way to determine at run time whether a pointer points to an allocated object of the appropriate type. Previous systems, such as CCured [24], addressed this problem by replacing pointers with multi-word pointers, known as "fat" pointers, that indicate the bounds for the pointer in question. Unfortunately, this approach changes the layout of data in the program, making it very difficult to "cure" one module or extension independently of the rest of the system.

In contrast, Deputy's types allow the programmer to specify pointer bounds in terms of other variables or structure fields in the program. Deputy's annotations also allow the programmer to identify null-terminated arrays and tagged unions. These annotations are flexible enough to accommodate a wide range of existing strategies for tracking pointer bounds. The key insight is that *most pointers' bounds are present in the program in some form—just not a form that is obvious to the compiler*. By adding appropriate annotations, we allow the compiler to insert memory safety checks throughout the program without changing the layout of the program's data structures.

Deputy is implemented as a source-to-source transformation that runs immediately after preprocessing. It is built on top of the CIL compiler framework [25], and the current prototype contains about 20,000 lines of OCaml code. Deputy has three phases:

1. *Inference.* First, Deputy infers bounds annotations for every unannotated pointer in the program. For unannotated local variables, Deputy inserts an annotation that refers to new local variables that explicitly track the unannotated pointer's bounds; this approach is essentially a variant of the fat pointer approach. For globals, structure fields, and function parameters and results, Deputy assumes a default pointer type. We use the inference module from CCured [24] to improve local variable annotations and to identify global variables that may require manual annotation.
2. *Type Checking.* Once all pointers have Deputy annotations, Deputy checks the code itself. It emits errors and run-time checks where appropriate.
3. *Optimization.* Since the type checking phase generates a large number of run-time checks, a Deputy-

specific optimization phase is used to eliminate checks that will never fail at run-time and to identify checks that will definitely fail at run time.

In the remainder of this section, we will discuss Deputy's types and run-time checks in more detail.

3.1 Deputy Type Annotations

In this section, we discuss Deputy's type annotations and their associated run-time checks. These type annotations allow Deputy to verify code that uses unsafe C features, such as pointer arithmetic, null-terminated strings, and union types. We focus on the informal semantics of these annotations; the technical details of the Deputy type system and its soundness argument are beyond the scope of this paper.

The annotations described in this section were chosen because they represent a reasonably large range of common C programming practices. They are simple enough to allow the type system to reason about them effectively, and yet they are expressive enough to be usable in real-world C programs.

It is important to note that the type annotations described in this section are *not* trusted by the compiler. Deputy checks when assigning a value to a variable that the value has the appropriate type; when using a variable, the type checker assumes that it contains a value of the declared type. In other words, these annotations function much like the underlying C types. Of course, Deputy only checks one compilation unit at a time, and therefore it must assume that other compilation units adhere to the restrictions of any Deputy annotations on global variables and functions, even if those other compilation units are not compiled by Deputy.

Buffers. Deputy allows the user to specify the bounds of a pointer by using one of four type annotations: `safe`, `sentinel`, `count(n)`, and `bound(lo, hi)`. In these annotations, `n`, `lo`, and `hi` stand for expressions that can refer to other variable or field names in the immediately enclosing scope. For example, annotations on local variables can refer to other local variables in the same function, and annotations on structure fields can refer to other fields of the same structure. These annotations can be written after any pointer type in the program; for example, a variable named `buf` could be declared with the syntax `int * count(len) buf`, which means that the variable `len` holds the number of elements in `buf`. The meanings of these annotations are as follows:

- The `safe` annotation indicates that the pointer is either null or points to a single element of the base type. Such pointers are the most common kind of pointer in C programs, and they typically require only a null check at dereference.
- The `sentinel` annotation indicates that a pointer is useful only for comparisons and not for dereference. This annotation is typically used for pointers that point immediately after an allocated area, as permitted by the ANSI C standard.
- The `count(n)` annotation indicates that the pointer is either null or points to an array of at least `n` elements. When accessing an element of this array, Deputy will insert a run-time check verifying that the pointer is non-null and the index is between zero and `n`.
- The `bound(lo, hi)` annotation indicates that the pointer is either null or points into an array with lower and upper bounds `lo` and `hi`. When accessing this array or applying pointer arithmetic to this pointer, Deputy will verify at run time that the pointer remains within the stated bounds.

Deputy allows casts between pointers with these annotations, inserting run-time checks as appropriate. For example, when casting a `count(n)` pointer to a `safe` pointer, Deputy will check that `n >= 1`. Similarly, when casting a pointer `p` with annotation `count(n)` to a pointer with annotation `bound(lo, hi)`, Deputy will verify that `lo <= p` and that `p + n <= hi`.

The fourth annotation, `bound(lo, hi)`, is quite general and can be used to describe a wide range of pointer bound invariants. For example, if `p` points to an array whose upper bound is given by a sentinel pointer `e`, we could annotate `p` with `bound(p, e)`, which indicates that `p` points to an area bounded by itself and `e`. In fact, the `sentinel`, `safe`, and `count(n)` annotations are actually special cases of `bound(lo, hi)`; when used on a pointer named `p`, they are equivalent to `bound(p, p)`, `bound(p, p + 1)`, and `bound(p, p + n)`, respectively. Thus, when checking these annotations, it suffices to consider the `bound(lo, hi)` annotation alone.

The invariant maintained by Deputy is as follows. At run time, any pointer whose type is annotated with `bound(lo, hi)` must either be null or have a value between `lo` and `hi`, inclusive. That is, for a pointer `p` of this type, we require that `p == 0 || (lo <= p && p <= hi)`. (Note that ANSI C allows `p == hi` as long as `p` is not dereferenced.) Furthermore, all of these pointers must be aligned properly with respect to the base type of this pointer. Given this invariant, we can verify the safety of a dereference operation by checking that `p != 0 && p < hi`. In order to ensure that this invariant is maintained throughout the program's execution, Deputy inserts run-time checks before any operation that could potentially break this invariant, which in-

```
int * safe find(int * count(len) buf,
                int len) {
  assert(buf != 0);
  int * sentinel end = buf + len;
  int * bound(cur, end) cur = buf;
  while (cur < end) {
    assert(cur != 0 && cur < end);
    if (*cur == 0) return cur;
    cur++;
  }
  return NULL;
}
```

Figure 4: Code showing the usage of Deputy bounds annotations. Underlined code indicates programmer-inserted annotations; boldface code indicates checks performed by Deputy at run time.

cludes changing the pointer itself or any other variable that appears in `lo` or `hi`. Since this process generates a large number of run-time checks, many of which are trivial (e.g., `p <= p`), Deputy provides an optimizer that is specifically designed to remove the statically verifiable checks that were generated by this process.

Figure 4 presents an example of Deputy-annotated code. This function finds and returns a pointer to the first null element in an array, if one exists. This example shows several annotations at work. The first argument, `buf`, is annotated with `count(len)`, which indicates that `len` stores the length of this buffer. The return type of this function, `int * safe`, indicates that we return a pointer to a single element (or null). (This annotation is not strictly necessary since `safe` is the default annotation on most unannotated pointers.) In the body of the function, we use a sentinel type for `end`, indicating that it cannot be dereferenced. Also, we use `cur` and `end` for the bounds of `cur`, which allows us to increment `cur` until it reaches `end` without breaking `cur`'s bounds invariant.

The boldface code in Figure 4 indicates the checks that were inserted automatically by Deputy based on the programmer-supplied annotations. When applying arithmetic to `buf`, we verify that `buf` is not null, since Deputy disallows arithmetic on null pointers. When dereferencing, incrementing, or returning `cur`, we verify that there is at least one element remaining in the array, which is a requirement for all of these operations. In many cases, Deputy's optimizer has removed checks that it determined to be unnecessary. For example, the result of `buf + len` is required to stay in bounds, so Deputy checks that `buf <= buf + len <= buf + len`; however, since `len` is known to be the length of `buf`, Deputy's optimizer has removed this check. In addition, the `cur++` operation requires the same check as the dereference in the previous statement; in this example, Deputy's optimizer has removed the duplicate check. Note that there are still many opportunities for further optimization of these checks; for example, a flow-sensitive optimizer could eliminate the second assertion entirely. We leave such improvements to future work, though it is worth noting that improvements in Deputy's run-time overhead are still well within reach.

```
size_t strlcpy(
      char * count(size-1) nullterm dst,
      const char * count(0) nullterm src,
      size_t size);
```

Figure 5: Example of Deputy's `nullterm` annotation.

Null termination. In addition to the basic pointer bounds annotations, Deputy also provides a `nullterm` annotation that can be used in conjunction with any one of the above annotations. This annotation indicates that the elements *beyond* the upper bound described by the bound annotation are a null-terminated sequence; that is, the bound annotation describes a subset of a larger null-terminated sequence. For example, `count(5) nullterm` indicates that a pointer points to an array of five elements followed by a null-terminated sequence. Note that `count(0) nullterm` indicates that a pointer points to an empty array followed by a null-terminated sequence—that is, it is a standard null-terminated sequence.

To see why this annotation is useful, consider the declaration for the `strlcpy()` function shown in Figure 5. The argument `dst` is annotated as `char * count(size-1) nullterm`, meaning that it has an least `size-1` bytes of real data followed by a null-terminated area (typically just a zero byte).[1] The `src` argument is `const char * count(0) nullterm`, which just means that it is a standard null-terminated string with unknown minimum length. These annotations demonstrate the flexibility of the `nullterm` annotation, since Deputy is capable of describing both a standard null-terminated pointer as well as a null-terminated array with some known minimum size.

The checks inserted for null-terminated sequences are a straightforward extension of the checks for bounds annotations. For example, when dereferencing a null-terminated pointer, we verify only that the pointer is non-null. When incrementing a null-terminated pointer, we check that the pointer stays within the declared bounds, and if not, we check that it is not incremented past the null element. Note that the `nullterm` annotation can be safely dropped during a cast, but it cannot be safely added to a pointer that is not null-terminated.

```
struct e1000_option {
  enum {range_option, list_option} type;
  union {
    struct {
        int min, max;
    } range when(type == range_option);
    struct {
        int nr;
        struct e1000_opt_list {...} *p;
    } list  when(type == list_option);
  } arg;
};
```

Figure 6: Code showing usage of Deputy's `when` annotation, adapted from Linux's `e1000` driver.

Tagged unions. Deputy also provides support for tagged unions. From the perspective of memory safety, C unions are a form of cast, allowing data of one type to be reinterpreted as data of another type if used improperly. C programmers usually use a tag field in the enclosing structure to determine which field of the union is currently in use, but the compiler cannot verify proper use of this tag. As with pointer bounds, Deputy allows the programmer to declare the conditions under which each field of the union is used so that these conditions can be verified at run time when one of the union's fields is accessed. To do so, the programmer adds the annotation "`when(p)`" to each field of the union, where `p` is a predicate that can refer to variable or field names in the immediately enclosing scope, and can use arbitrary side-effect-free arithmetic and logical operators.

For example, Figure 6 shows the annotations used by the Linux device driver `e1000`. In this example, the `type` field indicates which field of the union is currently selected. When using Deputy, the programmer can place the `when` annotations to indicate the correspondence between the tag and the choice of union field so that it can be checked when the union is accessed. For example, when reading the field `range`, Deputy will check that `type == range_option`. When the user wishes to change the tag field (or any field on which the `when` predicates depend), Deputy verifies that pointers in the newly selected union field are null and therefore valid.

There are two important restrictions that Deputy imposes on the use of tagged unions. First, they must be embedded in structures that contain one or more fields that can be used as tags for the union. Second, one cannot take the address of a union field, although it is possible to take the address of the structure in which it is embedded.

Other C features. Deputy supports a number of other common C features that we do not discuss in detail in this paper. For example, Deputy will use format strings to determine the types of the arguments in `printf`-style functions. Also, Deputy allows the program to annotate open arrays (i.e., structures that end with a variable-length array whose length is stored in another field of the structure). Finally, Deputy provides special annotations for functions like `memcpy()` and `memset()`, which require additional checks beyond those used for the core Deputy annotations.

3.2 Annotating Programs

The Deputy type system expects to see a pointer bound annotation on every pointer variable in the program in order to insert checks. However, since adding annotations to every pointer type would be difficult and would clutter the code, Deputy provides a simple inference mechanism. For any local variable of pointer type and for any cast to a pointer type, Deputy will insert two new variables that explicitly hold the bounds for this type. The type can then be automatically annotated with `bound(b, e)`, where `b` and `e` are the two new variables. Whenever this variable is updated, Deputy inserts code to update `b` and `e` to hold the bounds of the right-hand side of the assignment. Although this instrumentation inserts many additional assignments and variables, the trivial ones will be eliminated by the Deputy-specific optimizer.

We also use the inference algorithm from CCured [24] to avoid inserting unnecessary local variables. For example, if a local variable is incremented but not decremented, we insert a new variable for the upper bound only. This inference algorithm is also useful for inferring `nullterm` annotations.

For pointer types other than those found in local variables or casts, Deputy requires the programmer to insert an annotation. Such pointer types include function prototypes, structure fields, and global variables. These annotations are required in order to support separate compilation; since Deputy cannot instrument external function arguments or structures fields the same way it can instrument local variables, the programmer must explicitly supply bounds annotations. Fortunately, most of these annotations appear in header files that are shared among many compilation units, which means that each additional annotation will benefit a large number of modules in the program. Also, Deputy can optionally use a default value for unannotated pointers in the interface (usually `safe`); however, such annotations are unreliable and should eventually be replaced by an explicit annotation. The inference algorithm assists in this process as well by identifying global variables, functions, and structure fields that require explicit annotations.

3.3 Limitations and Future Work

The most significant safety issue that is ignored by Deputy is the issue of memory deallocation. Deputy is designed to detect safety violations due to bounds violations, but it does not check for dangling pointers, since detecting such violations would require more extensive run-time checking or run-time support than we currently provide. Fortunately, the issue of dangling pointers is largely orthogonal to the problems that Deputy solves; thus, the programmer can choose to use a conservative garbage collector or to simply trust that the program's deallocation behavior is correct.

In addition, there are some cases in which Deputy's type system is insufficient for a given program. First, Deputy's type system is sometimes incapable of expressing an existing dependency; for example, programmers sometimes store the length of an array in a structure that is separate from the array itself. Second, Deputy's type system sometimes fails to understand a type cast where there are significant differences between the pointer's base type before and after the cast. Based on our experience with device drivers (see Section 5) and our previous experience with CCured [24], such casts occur at about 1% of the assignments in a C program. In both of these cases, Deputy provides a trusted cast mechanism that allows the programmer to suppress any Deputy errors or run-time checks for a particular cast or assignment, effectively flagging this location for more detailed programmer review. One of our primary goals in designing Deputy and instrumenting programs is to minimize the number of such casts required; however, in many of these cases, a failure in Deputy's type system indicates a lack of robustness in the code itself. Thus, the process of annotating code can help identify places where the interface between modules can be improved.

Deputy does not provide any special handling for multithreaded code, since correctly synchronized code will still be correctly synchronized after being compiled with Deputy. However, C code that was previously atomic may no longer be atomic due to the addition of run-time checks. Thus, the programmer must ensure that their code makes no unwarranted assumptions about the atomicity of any particular piece of C code.

Finally, Deputy's changes to the source code (e.g., temporary variables and run-time checks) can make source-level debugging more challenging. However, with sufficient engineering effort, code compiled with Deputy could be viewed in the debugger with the same ease as code compiled by GCC. If necessary, code can currently be compiled and debugged with GCC first, using Deputy only for the final stages of development.

In the future, we hope to provide support for many of the programming idioms that currently require trusted casts. For example, the Linux `container_of` macro, which subtracts from a pointer to a field to get a pointer to the containing object, is a construct we hope to support in a future version. We also plan to improve our handling of `void*` pointers, particularly for `void*` pointers that appear in kernel data structures in order to hold private data for a module. And finally, we plan to improve Deputy's optimizer in order to further reduce the number of run-time checks required. Our long-term goal is to allow programmers to migrate drivers (and even the kernel itself) to a type-safe version of C without significantly rewriting their code or affecting performance.

4 Recovery System

This section describes how SafeDrive tracks updates from the driver and recovers from driver failures. We also compare our recovery mechanisms to that of Nooks.

4.1 Update Tracking

The update tracking module maintains a linked list of all updates a driver has made to kernel state that should be undone if the driver fails. For each update, the list stores a *compensation operation* [36], which is a pointer to a function that can undo the original update, along with any data needed by this compensating action. For example, in Linux device drivers, the compensation function for `kmalloc()` is `kfree()`. This list is also indexed by a hash table, which allows compensations to be removed from the list if the driver manually reverses the update (e.g., if an allocated block is freed). SafeDrive provides wrappers for all functions in the kernel API that require update tracking, allowing drivers to use this feature with minimal changes to their source code.

In a few cases, we need to modify the kernel to handle changes to the list of updates that are not explicitly performed by the driver. For example, timers are removed from the list after the corresponding timer function executes.

Updates recorded by the tracking module are divided into two separate pools, one associated with the driver itself (long-term updates) and the other associated with the current CPU and control path (short-term updates). The latter pool holds updates like spinlock-acquires, which have state associated with the local CPU and must be undone atomically (without blocking) on the same CPU.

4.2 Failure Handling

Failures are detected by the run-time checks inserted by the Deputy compiler. When a run-time checks fails, it invokes the SafeDrive recovery system. First, a description of the error is printed for debugging purposes. For

problems due to memory safety bugs, this error report pinpoints the actual location where a pointer leaves its designated bounds or is dereferenced inappropriately.

Then SafeDrive goes through a series of steps to clean up the driver module itself, restoring kernel state and optionally restarting the driver, while at the same time allowing other parts of the system to continue running. We maintain two invariants during the recovery process, both vital to the success of recovery:

- *Invariant 1: No driver code is executed from the point when a failure is detected until recovery is complete*. This invariant is required because the driver is already corrupt; executing driver code could easily trigger more failures or corrupt kernel state.
- *Invariant 2: No kernel function is forcefully stopped and returned early.* Forceful returns from kernel functions would corrupt kernel state. On the other hand, the driver function that fails is always stopped forcefully, in order to maintain the first invariant.

Returning gracefully from a failed driver. The basic mechanism for "forceful return" from a driver function is a `setjmp()/longjump()` variant that unwinds the stack and jumps directly to the next instruction after `setjmp()`. SafeDrive requires the programmer to identify driver entry points, and at these entry points, SafeDrive adds a standard wrapper stub that calls `setjmp()` to store the context in the current task structure (Linux's kernel thread data structure). When a failure occurs, the failure-handling code will call `longjmp()` to return to the wrapper, which then returns control to the kernel with a pre-specified return value, often indicating a temporary error or busy condition. The failure-handling code also sets a flag that indicates that the driver is in the "failed" state. This flag is checked at each wrapper stub to ensure that any future calls to the driver will return immediately, thus preserving Invariant 1. This approach allows the kernel to continue normally when the driver fails.

Typically, identifying driver entry points is a simple task; to a first approximation, we can look for all driver functions whose address is taken. Determining the appropriate return value for a failure can be more difficult, since returning an inappropriate value can cause the kernel to hang. Fortunately, the appropriate return value is relatively consistent across each class of drivers in Linux. The Nooks authors provide a more extensive study of possible strategies for selecting proper return values [32].

The recovery process is more complicated in cases where the driver calls back into the kernel, which then calls back into the driver, resulting in a stack that contains interleaved kernel and driver stack frames. If we jump to the initial driver entry point, we skip important kernel code in the interleaved stack frame, violating Invariant 2. However, if we jump to the closest kernel stack frame, we must ensure that Invariant 1 is maintained when we return to the failed driver. To solve this problem, SafeDrive records a context at each re-entry point into the driver. A counter tracks the level of re-entries into the driver, which is incremented whenever the driver calls a kernel function that *may* call back into the driver. After the driver fails, control jumps directly to re-entry points when it returns from these kernel functions. Essentially, we finish executing any kernel code still on the stack while skipping any driver code still on the stack. This technique is similar to the handling of domain termination in Lightweight RPCs [4], although in our case, both the kernel and the module run in the same protection domain.

Restoring kernel state and restarting driver. During the recovery process, all updates associated with the failed driver are undone in LIFO order by calling the stored compensation functions. These compensations undo all state changes the driver has made to the kernel so far, similar to exception handling in languages like C++ and Java. The main difference between our approach and C++/Java exceptions is that the compensation code does not contain any code from the extension itself, thus preserving Invariant 1. As a result, the extension will not have an opportunity to restore any local invariants; however, because the extension will be completely restarted, we are only concerned with restoring the appropriate kernel invariants. Note that this unwinding task is complicated by the fact that it is executed in parallel with other kernel code and by the fact that the failure could have happened in an inconvenient place, such as an interrupt handler or timer callback. Thus, after CPU-local compensations such as lock releases are undone in the current context, all other compensations are deferred to be released in a separate kernel thread (in `event/*`).

After compensations have been performed, the driver's module is unloaded. As mentioned above, we do not call the driver's deinitialization function; however, because we track all state-changing functions provided by the kernel, including any function that registers new devices or other services, calling the driver's deinitialization function should not be necessary. After this process is complete, depending on user settings, the driver can be restarted automatically from a clean slate. The restart process follows the normal module initialization process.

The current SafeDrive update tracking and recovery code is a patch to the Linux kernel changing 1084 lines, tracking in total 21 types of Linux kernel resources for the recovery of four drivers in three classes: two network card drivers, one sound card driver, and one USB device

driver (see Section 5 for details). The resources tracked include 4 types of locks, 4 PCI-related resources, and 4 network-related resources, among others.

4.3 Discussion

The recovery system of SafeDrive resembles that of Nooks [33]. Both track different types of kernel updates and undo them at failure time. However, the fact that all code using SafeDrive runs in the same protection domain makes SafeDrive's recovery system significantly simpler and less intrusive. In fact, the SafeDrive kernel patch is less than one tenth the size of Nooks'. Update-tracking wrappers and compensation functions are simpler mainly because there is no need for code to copy objects in and out of drivers, to manage the life cycles of separate protection domains, or to perform cross-domain calls. In addition, implementing the cross-domain calls efficiently can complicate the system design significantly. For example, Nooks uses complicated page table tricks to enable fast writes of large amount of data from the device driver to the kernel. Nooks also gives each domain its own memory allocation pool, which requires even more changes to the kernel. We believe that simpler recovery code adds significantly to the trustworthiness of the whole mechanism since testing is less effective for such code than for the normal execution path.

The fact that SafeDrive lets drivers modify kernel data structures directly has some ramifications for kernel consistency, which contrasts with Nooks' call-by-value-result object passing. For each driver call, Nooks first copies all objects the driver may modify, then lets the driver work on the copies, and finally copies the results back. This approach provides atomic updates of kernel objects and thus should provide more consistency. However, we have found there are caveats to this approach. First, it has problems on SMP systems because other processors see the updates to kernel data structures at a different time than they would ordinarily see the updates; the original Nooks paper suggests that SMP is not supported yet [33, end of Sec. 4.1]. Second, many kernel data structure updates are not done through this mechanism but through cross-domain calls to kernel functions. The interaction of these two mechanisms becomes complicated quickly.

Our recovery scheme is orthogonal to the Deputy type system and can also work with other languages, including type-safe languages such as Java. As far as we know, current type-safe languages and runtimes do not provide a general mechanism for recovering buggy extensions. Our technique for gracefully returning from a failed extension should apply here. More broadly, a recovery module based on compensation stacks [36] would be a valuable asset for these systems.

Driver	Description
e1000	Intel PRO/1000 network card driver
tg3	Broadcom Tigon3 network card driver
usb-storage	USB mass-storage driver
intel8x0	Intel 8x0 builtin sound driver
emu10k1	Creative Audigy 2 sound driver
nvidia	NVidia video driver

Table 1: Drivers used in experiments.

State and session restoration for failed drivers is not yet implemented in our prototype. Thus, the driver will be in an initial empty state after recovery. However, we believe the shadow driver approach proposed by the Nooks project [32] should apply to our system. Its implementation should also be simpler because of the absence of multiple hardware protection domains.

5 Evaluation

In this section, the recovery mechanism is first evaluated in terms of successful recoveries in the face of randomly injected faults. Then we measure two distinct types of overhead of using SafeDrive: (1) the one-time overhead of annotating APIs and adding wrapper functions to a particular extension, and (2) runtime overhead due to additional checks inserted by the Deputy type system and due to update tracking for recovery. We quantify the first one by noting how much of the kernel API and wrappers needed alteration to support a handful of drivers that we chose from different subsystems of the kernel. We quantify the second source of overhead with traditional performance benchmarks.

Our experiments are done with six device drivers in four categories for the Linux 2.6.15.5 kernel, as shown in Table 1. All drivers are annotated with Deputy annotations to detect type-safety errors. However, due to time constraints, we only added update tracking and recovery support to the four drivers with names shown in bold.

5.1 Recovery Rate

Here we evaluate how well SafeDrive is able to handle various faults in drivers. We use *compile-time software-implemented fault injection* to inject random faults into the driver, and then we run the driver with and without SafeDrive. We wrote a compile-time fault injection tool as an optional phase in the Deputy compiler. The tool injects into C code seven categories of faults, shown in Table 2, following two empirical studies on kernel code [10, 31]. We did not use an existing binary-level fault injection tool, such as that used by the Rio file cache [26] or Nooks [34], because all of Deputy's checks are inserted at compilation time, which means that Deputy

Fault Category	Code Transformation
Loop fault	Make loop count larger by: 1 with 0.5 prob., 2–1024 with 0.44 prob., and 2K–4K with 0.06 prob.
Scan overrun	Make size parameter to `memset`-like functions larger as the line above
Off-by-one	Change $<$ to $<=$, etc., in boolean expressions
Flipped condition	Negate conditions in if statements
Missing assignment	Remove assignments and initialization of local vars
Corrupt parameter	Replace a pointer parameter with null, or a numeric parameter with a random number
Missing call	Remove calls to functions and return a random result as in the line above

Table 2: Categories of faults injected in recovery experiments.

	Correct		Incorrect	
SafeDrive	Works	Innocuous Errors	Crashes	Mal-functions
Off	75	*n/a*	44	21
On	113	8	0	19

Table 3: Results of 140 runs of `e1000` with injected faults, with SafeDrive off and on. "Correct" means that the driver behaved as expected, possibly with the help of SafeDrive's recovery subsystem and assuming the bugs are transient.

Detection	Crashes	Malfunc.	Innocuous	Total
Static	10	0	3	13 (24%)
Dynamic	34	2	5	41 (76%)
Total	44	2	8	54

Table 4: Breakdown for the 54 cases in which SafeDrive detected errors.

does not have a chance to catch errors introduced via binary fault injection. Compile-time fault injection allows us to evaluate Deputy's error detection fairly, and it also has the benefit of being able to introduce more realistic programming errors.

The fault-injection experiments were done with `e1000` driver. For each experiment, 5 random faults of the same category were inserted into the driver code during the compilation process. A script exercised the driver by loading it, configuring networking, downloading a large (89MB) file, checksumming the file, and finally unloading the driver. Then the same script was run with the unmodified `e1000` driver to check whether the system was still functioning. This test was performed with and without SafeDrive recovery on. When SafeDrive recovery was off, Deputy checks were still performed, but they did not trigger any action when failures were detected. Thus the driver should have behaved exactly the same as the original one with faults injected.

Table 3 shows the results of all 140 runs, with 20 runs per fault category. When SafeDrive is disabled, 44 of these runs resulted in crashes and 21 resulted in the driver malfunctioning. When SafeDrive was enabled, SafeDrive successfully prevented all 44 crashes, and it invoked the recovery subsystem on 2 of the 21 non-crash driver malfunctions. In addition, there were 8 runs where SafeDrive successfully detected apparently innocuous type-safety errors that did not trigger crashes or malfunctions with SafeDrive disabled, but failed Deputy checks when SafeDrive was on.

A closer look at the results reveals that, among the 7 categories of faults injected, all categories except "off-by-one" and "flipped condition" resulted in crashes when SafeDrive was off. The fact that SafeDrive prevented these crashes indicates that these crashes were actually due to type-safety errors, not other serious errors such as incorrect interrupt commands. On the other hand, the malfunctioning runs were due to a variety of reasons, including setting hardware registers to bad values and incorrect return value checking. Type-safety checks alone cannot always detect these errors.

Overall, SafeDrive detected problems in 54 of the runs. Table 4 shows that 24% of the problems were detected statically by the Deputy compiler. These are errors that were *not* detected by GCC, and in fact 10 of them led to crashes. These compile-time errors include passing an incorrect constant size argument to `memcpy` and dereferencing an uninitialized pointer, among others.

Of the innocuous errors, five were detected dynamically, causing SafeDrive to invoke recovery and successfully restart the driver. Although these errors appear to be innocuous, they are likely latent bugs; thus, invoking recovery seems to be a reasonable response. Of course, SafeDrive has no way to know which errors will be truly benign.

These results show that SafeDrive is effective in detecting and recovering from typical memory and type-safety errors in drivers. In addition, a significant portion of these errors are caught at compile time, before the driver is even run. Finally, it seems unlikely that SafeDrive will always prevent *all* crashes, as it did with the `e1000`, due to the limits of type checking.

5.2 Annotation Burden

Table 5 shows the number of lines of code we changed in order to use SafeDrive on these drivers. The third column shows all Deputy-related changes, and the next four columns show the number of lines containing each type of Deputy annotation. These numbers do not add up to

Driver	Original LOC	Deputy Changes					Recovery Changes
		LOC Modified	Bounds	Nullterm	Tagged Unions	Trusted Code	
e1000	17011	260	146	15	2	47	270
tg3	13270	359	78	9	0	64	156
usb-storage	13252	136	16	11	0	21	118
intel8x0	2897	124	31	2	0	8	167
emu10k1	11080	441	66	11	0	23	n/a
nvidia	10126	224	42	35	0	27	n/a

Table 5: Number of lines changed in order to enable SafeDrive for each driver. All numbers are lines of code. "LOC Modified" in "Deputy Changes" are number of lines with Deputy annotations and related changes. The next four columns show numbers of lines with each category of annotations. "Recovery Changes" shows the number of lines where trivial wrappers were added for recovery.

the previous number because not every changed line contains a Deputy annotation. These changes are essential to driver safety, and they amount to approximately 1–4% of the total number of lines in a given driver. The last column shows additional recovery-related changes currently needed in the driver code by the SafeDrive prototype. These changes are boilerplate code placed at driver entry points in the driver source files. We intend to generate these wrappers automatically in the future.

The other kind of changes required are changes to the kernel headers. This set of changes includes Deputy annotations for the kernel API as well as wrappers for functions that are tracked by the SafeDrive runtime system. Both types of changes must be written by hand; however, these changes are a one-time cost for drivers of a given class. For the 4 classes of drivers we worked on, a total of 1,866 lines in 112 header files were changed. Casual inspection reveals that about half of the lines are Deputy-related and that the rest are recovery-related. In particular, there are 187 lines with bounds annotations, 260 lines with nullterm, 8 lines with tagged unions, and 140 lines with trusted code annotations.

5.3 Performance

The run-time overhead of SafeDrive is composed of several parts, including run-time checks inserted by the Deputy compiler, update tracking cost, and context saving cost. We measure the performance overhead of SafeDrive by running several benchmarks with native and SafeDrive-enabled drivers.

Table 6 shows results for these benchmarks on a dual Xeon 2.4Ghz. In "TCP Receive", `netperf` [18] is run on another host and sends TCP streaming data to the testing server (the `TCP_STREAM` test). The socket buffers are 256KB on both the sending and receiving side, and 32KB messages are sent. "TCP Send" works the other way around, with the testing server running `netperf` and sending traffic. In "UDP Receive" and "UDP Send", UDP packets of 16 bytes are received/sent (the `UDP_STREAM` test). CPU utilization is measured with the `sar` utility. CPU utilization maxes out at 50%, probably because the driver cannot utilize more than one CPU. The UDP tests show higher overhead probably due to two reasons. First, the packets are much smaller, leading to more Deputy overhead overall. Second, less other kernel code is involved in UDP processing compared to TCP, amplifying SafeDrive's overhead.

We tested the `usb-storage` driver with a 256MB Sandisk USB2.0 Flash drive on a Thinkpad T43p laptop (2.13Ghz Pentium-M CPU). The "Untar" benchmark simply untars a Linux 2.2.26 source code tar ball, which is already on the drive, to the drive itself. The tar file is 82MB in size. After untarring finishes, the drive is immediately unmounted to flush any data in the page cache. CPU utilization is the average value over the whole period. As can be seen from the results, the whole operation finishes in the same amount of time, though SafeDrive's instrumentation increased CPU usage by 23%.

We also benchmarked two sound card drivers: the `intel8x0` sound driver for the built-in sound chip in a Thinkpad T41 (1600Mhz Pentium-M CPU), and the `emu10k1` sound driver for a Creative Audigy 2 card on a Pentium II 450Mhz PC. Both benchmarks used the `oprofile` facility to capture how much time (in percentage of total CPU time) was spent in the kernel on behalf of the sound driver while a 30-minute 44.1Khz wave file was playing. `aplay` and the standard `alsa` sound library were used for playback. Throughput is irrelevant here because the sample rate is fixed.

Finally, we tested the open-source portion of the driver distributed by NVidia for their video cards. These 10,296 lines of open-source code are the interface between the kernel and a larger, proprietary graphics module which we did not process because the source code is not available. We tested the `nvidia` driver on a Pentium 4 2.4 GHz machine with a GeForce4 Ti 4200 graphics card. Table 6 shows the CPU usage of this driver while setting up and tearing down an X Window session, as measured by `oprofile`. The instrumentation did not have a measurable effect on the performance of the `x11perf` graphics benchmarking tool, which is limited by hard-

Benchmark	Driver	Native Throughput	SafeDrive Throughput	Native CPU %	SafeDrive CPU %
TCP Receive	e1000	936Mb/s	936Mb/s	47.2	49.1 (+4%)
UDP Receive	e1000	20.9Mb/s	17.4Mb/s (-17%)	50.0	50.0
TCP Send	e1000	936Mb/s	936Mb/s	20.1	22.5 (+12%)
UDP Send	e1000	33.7Mb/s	30.0Mb/s (-11%)	45.5	50.0 (+9%)
TCP Receive	tg3	917Mb/s	905Mb/s (-1.3%)	25.4	27.4 (+8%)
TCP Send	tg3	913Mb/s	903Mb/s (-1.1%)	18.0	20.4 (+13%)
Untar	usb-storage	1.64MB/s	1.64MB/s	5.5	6.8 (+23%)
Aplay	emu10k1	n/a	n/a	9.10	9.64 (+6%)
Aplay	intel8x0	n/a	n/a	3.79	4.33 (+14%)
Xinit	nvidia	n/a	n/a	12.13	12.59 (+4%)

Table 6: Benchmarks measuring SafeDrive overhead. Utilization numbers are kernel CPU utilization.

ware performance rather than by the driver.

The above results show that SafeDrive has relatively low performance overhead, even for data-intensive drivers. To compare with Nooks, consider the `e1000` TCP send/receive tests. These tests are similar to experiments discussed in the Nooks journal paper [34], where the authors reported relative receive and send CPU overheads of 111% and 46% respectively, both with 3% degradation of throughput. This overhead is nearly an order of magnitude higher than the overhead of SafeDrive (rows 1 and 3 in Table 6), suggesting that the overhead of cross-domain calls far outweighs the overhead of Deputy's run-time checks for these benchmarks.

6 Related Work

6.1 Enforcing Isolation with Hardware

Several projects have used hardware to isolate device drivers from the rest of the system and to allow recovery in the case of failure.

The Nooks project [32, 33] isolates device drivers from the main kernel by placing them in separate hardware protection domains called "nooks". These protection domains share the same address space but have different permission settings for pages. A driver is permitted to read all kernel data but only allowed to write to certain pages. Cross-domain calls replace function calls between the driver and the kernel, although the semantics of the calls remain mostly similar.

Virtual Machine Monitors (VMMs) such as L4 [21] and Xen [3, 13] isolate a driver using hardware protection. However, rather than placing a protection barrier between a driver and the kernel, the VMM runs each driver with its own kernel inside a separate virtual machine. This approach allows device drivers to be used unchanged, and it allows one to use device drivers that depend on different operating systems. Communication between virtual machines is performed using a special-purpose high-performance batched channel interface.

From a pure driver isolation standpoint, SafeDrive is less secure than a VMM since it is possible for a driver to manipulate the kernel via the API calls, privileged instructions, or simply tying up CPU resources by looping. In the VMM case, however, a buggy driver can only corrupt the driver's local, untrusted kernel, which is isolated from the trusted kernel behind a message-passing interface. The SLAM project [2] looks at API usage validation, and in theory could close this gap when combined with SafeDrive.

However, this caveat does not mean that SafeDrive is able to catch fewer errors than a VMM. In fact, one advantage of SafeDrive is that its finer-grained checks are able to detect errors within the driver and not just outside it. Although hard to measure, this fine-grained error detection reduces the likelihood of data corruption and is very important in cases involving persistent data. For example, a misbehaving disk driver in any of Xen, Nooks, or SFI can corrupt data on disk before the fault is detected. This exact behavior occurred with Nooks during their fault injection experiments [33].

Another advantage of SafeDrive relative to these systems is performance. The additional domain crossings required by the hardware approaches impose additional costs. In all three hardware-based systems, one can generally expect the CPU overhead for data intensive device drivers to be between 40% to 200% [21, 22, 32, 34]. This result contrasts with a typical CPU overhead of less than 20% for SafeDrive, which incurs no additional cost for calls into or out of a driver. Of course, it is not guaranteed that SafeDrive will always outperform hardware approaches: if crossings are rare and checks are frequent then a hardware solution is likely to outperform SafeDrive; however, our experiments suggest that SafeDrive performs better in practice and that SafeDrive's performance is likely to improve further as its optimizer improves.

6.2 Enforcing Isolation with Binary Instrumentation

Software-enforced Fault Isolation (SFI) [11, 29, 35] instruments extension binaries to ensure that no memory operation can write outside of an extension's designated memory region. The instrumentation can take one of several forms, depending on the desired tradeoff between isolation and performance.

If reads are protected as well as writes (as they are in SafeDrive), then typical performance overhead varies between 17% and 144%, depending on the SFI implementation and the benchmarks being used. Although it is hard to make direct performance comparisons against results obtained from different test programs, we expect SafeDrive to exceed the performance of SFI, since SafeDrive only needs to check memory accesses for which the type checker is unable to verify correctness.

As with the hardware-based approaches, SFI only prevents an extension from corrupting the system, and it does not attempt to prevent a driver from corrupting itself or the device. Furthermore, these approaches require a very clean kernel-driver interface in order to ensure that all data passed between the kernel and the driver can be checked at run time [11].

6.3 Enforcing Isolation with a Language

A number of research projects have attempted to enforce memory safety in C programs at source level instead of at binary level. CCured [24], a predecessor to Deputy, used a whole-program analysis to classify pointers according to their use, and then it altered data structures and code to provide low-cost run-time memory safety checks. Unfortunately, CCured made significant changes to the program's data structures, making it difficult to apply CCured to one module at a time. Another source-level tool is the Cyclone [19] language, which is a safe alternative to the C language that has been used to write safe kernel-level code [1]. However, like CCured, Cyclone requires a large amount of manual intervention to port existing drivers when compared to Deputy. Neither CCured nor Cyclone support *a priori* data layouts, which are a prerequisite for extensions with predefined APIs and data structures. Finally, Yong and Horwitz [38] use static analysis to insert efficient buffer overflow checks; however, they do not address memory safety in general, and they provide relatively coarse-grained checks.

The major advantage of the Deputy type system over these other source-level approaches is that Deputy allows the programmer to describe pointer bounds in terms of other variables or fields in the program, and thus Deputy can leave data layout and APIs unchanged. A related project at Microsoft uses the SAL annotation language and the ESPX modular annotation checker in order to find buffer overflows [14]. Although SAL can describe relationships between variables, it cannot describe relationships between structure fields, and it does not support tagged union types. Also, ESPX is a static analysis, which means that problematic code is simply left unverified; in contrast, Deputy inserts run-time checks where static analysis is insufficient.

There are also a number of related projects that allow types to refer to program data, including the Xanadu language [37] and Hickey's very dependent function types [16]. However, these projects have a number of restrictions on mutable data, which Deputy addresses using run-time checks. Also, Harren and Necula [15] developed a similar framework for assembly language in which dependencies can occur between registers or between structure fields.

Type qualifiers represent another area of related work. CQual [12, 20] allows programmers to add custom type qualifiers such as `const` or `user`/`kernel`, using an inference algorithm to propagate these qualifiers throughout the program. Semantic type qualifiers [8] builds on this work by allowing qualifiers to be proved sound in isolation. Compared to this work, Deputy's annotations are more expressive (since annotations can refer to other program data) and correspondingly more difficult to infer. Although Deputy provides a number of features for inferring or guessing annotations, it relies more heavily on programmer-supplied annotations than these other type qualifier tools. Finally, Privtrans [6] allows the programmer to specify privileged operations and automatically separates a program into a privileged process and non-privileged process, improving security.

There are operating systems built mainly with type-safe languages, such as Singularity [17], JavaOS [23], and the Lisp Machine OS [30]. These operating systems naturally have few memory-safety problems, and as processor speed increases, the performance penalty of these languages become less of a concern. While this approach will be important in building future operating systems, we believe that current commodity operating systems such as Windows and Linux, which are written mainly in C and C++, will be in wide use for many years. SafeDrive will be useful both in improving the reliability of these existing operating systems and in providing a transition to future type-safe operating systems.

6.4 Fault Tolerance for Applications

In addition to the issue of how a device driver should be isolated from the rest of the system, there is also the largely orthogonal issue of how the system should recover when a driver failure is detected. Both the original Nooks paper [33] and SafeDrive provide isolation,

release of resources, and restart of the driver.

A later Nooks paper [32] showed how to use *shadow drivers* to restore the session state of applications that were using the driver. We expect the shadow driver technique to work without significant modification with SafeDrive, and thus we do not address this issue further.

Vino [28], which isolates drivers using SFI, executes each driver request inside a transaction that can be aborted and retried on failure. Another related technique is Microreboot [7], which is used to build rebootable components in large enterprise systems. Session restoration is achieved by programming the components against a separate session state storage that persists over component restarts.

7 Conclusion

We have presented SafeDrive, a system that uses language-based techniques to detect type safety errors and to recover from such errors in device drivers written in C. The checking in SafeDrive is fine-grained, which is critical because it not only protects the kernel from misbehaving drivers, but also helps prevent the driver from corrupting persistent data or kernel state. SafeDrive requires few changes to the kernel or drivers, and experiments show that SafeDrive incurs low overhead (normally less than 20%) and successfully prevents all 44 crashes due to randomly injected errors for one driver. Overall, we hope that this work shows that we can achieve the safety of high-level, type-safe languages without abandoning existing C code.

Acknowledgments

We thank the anonymous reviewers and our shepherd Dawn Song for their useful suggestions and comments on the paper. We also thank David Gay for insightful discussion.

Notes

[1]This requirement is slightly different from the official `strlcpy` specification, since `dst` is required to be null-terminated on entry as well as on exit. However, in practice, establishing this invariant on entry should not require much, if any, additional effort on the part of client code.

References

[1] ANAGNOSTAKIS, K., GREENWALD, M., IOANNIDIS, S., AND MILTCHEV, S. Open packet monitoring on FLAME: Safety, performance and applications. In *Proceedings of the 4rd International Working Conference on Active Networks (IWAN 2002)* (2002).

[2] BALL, T., MAJUMDAR, R., MILLSTEIN, T. D., AND RAJAMANI, S. K. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation* (2001).

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[4] BERSHAD, B., ANDERSON, T., LAZOWSKA, E., AND LEVY, H. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (1989).

[5] BOS, H., AND SAMWEL, B. Safe kernel programming in the OKE. In *Open Architectures and Network Programming Proceedings* (2002).

[6] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium* (2004).

[7] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot — a technique for cheap recovery. In *Symposium on Operating System Design and Implementation* (2004).

[8] CHIN, B., MARKSTRUM, S., AND MILLSTEIN, T. Semantic type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (2005).

[9] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (2001).

[10] CHRISTMANSSON, J., AND CHILLAREGE, R. Generation of an error set that emulates software faults — based on field data. In *Proceedings of the 26th IEEE International Symposium on Fault Tolerant Computing* (1996).

[11] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation* (2006).

[12] FOSTER, J. S., FAHNDRICH, M., AND AIKEN, A. A theory of type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (1999).

[13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND MARK WILLIAMSON. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004)* (2004).

[14] HACKETT, B., DAS, M., WANG, D., AND YANG, Z. Modular checking for buffer overflows in the large. Technical Report MSR-TR-2005-139, Microsoft Research, 2005.

[15] HARREN, M., AND NECULA, G. C. Using dependent types to certify the safety of assembly code. In *Procedings of the 12th international Static Analysis Symposium (SAS)* (2005).

[16] HICKEY, J. Formal objects in type theory using very dependent types. In *Proceedings of the 3rd International Workshop on Foundations of Object-Oriented Languages* (1996).

[17] HUNT, G., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. D. An overview of the Singularity project. Tech. rep., Microsoft Research, 2005.

[18] INFORMATION NETWORKS DIVISION, H.-P. C. Netperf: A network performance benchmark. http://www.netperf.org.

[19] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference* (2002).

[20] JOHNSON, R., AND WAGNER, D. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium* (2004).

[21] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symposium on Operating System Design and Implementation* (2004).

[22] MENON, A., SANTOS, J. R., TURNER, Y., G. (JOHN) JANAKIRAMAN, AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceeding of the 1st ACM/USENIX Conference on Virtual Execution Environments (VEE 2005)* (2005).

[23] MITCHELL, J. G. JavaOS: Back to the future (abstract). In *Symposium on Operating System Design and Implementation* (1996).

[24] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems 27*, 3 (2005).

[25] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CIL: Intermediate language and tools for the analysis of C programs. In *International Conference on Compiler Construction* (2002).

[26] NG, W. T., AND CHEN, P. M. The systematic improvement of fault tolerance in the Rio file cache. In *Symposium on Fault-Tolerant Computing* (1999).

[27] PATEL, P., WHITAKER, A., WETHERALL, D., LEPREAU, J., AND STACK, T. Upgrading transport protocols using untrusted mobile code. In *SOSP* (2003).

[28] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symposium on Operating System Design and Implementation* (1996).

[29] SMALL, C., AND SELTZER, M. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies (COOT 1997)* (1997).

[30] STALLMAN, R., WEINREB, D., AND MOON, D. *The Lisp Machine Manual*. Massachusetts Institute of Technology, 1981.

[31] SULLIVAN, M., AND CHILLAREGE, R. Software defects and their impact on system availability — a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing* (1991).

[32] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. In *Symposium on Operating System Design and Implementation* (2004).

[33] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[34] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Transactions Computer Systems 23*, 1 (2005).

[35] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (1993).

[36] WEIMER, W., AND NECULA, G. C. Finding and preventing run-time error handling mistakes. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004).

[37] XI, H. Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science* (2000).

[38] YONG, S. H., AND HORWITZ, S. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2003).