Inferring Data Polymorphism in Systems Code*

Brian Hackett
Stanford University
bhackett@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

We describe techniques for analyzing *data polymorphism* in C, and show that understanding data polymorphism is important for statically verifying type casts in the Linux kernel, where our techniques prove the safety of 75% of downcasts to structure types, out of a population of 28767. We also discuss prevalent patterns of data polymorphism in Linux, including code patterns we can handle and those we cannot.

General Terms

Verification, Experimentation

Keywords

type checking, type casting, static analysis

1. INTRODUCTION

Consider a typical Linux function, `saa7146_buffer_timeout`, which is part of the device driver for the saa7146 chipset:

```
// drivers/media/common/saa7146_fops.c
void saa7146_buffer_timeout(unsigned long data)
{
   struct saa7146_dmaqueue *q = (struct saa7146_dmaqueue*)data;
   struct saa7146_dev *dev = q->dev;
   unsigned long flags;
   ...
}
```

This function casts its integer parameter `data` to a pointer to type `saa7146_dmaqueue` and then accesses the contents of that structure. If `data` really is an integer, or if `data` is a pointer to an object that is not of type `saa7146_dmaqueue`, then these accesses will corrupt or crash the system. Type casts like this one are ubiquitous in Linux and other large C codebases. Analyzing these casts to determine their correctness requires deep reasoning about the heap, control flow and data flow of the system. In addition, many of these casts are used to implement polymorphism (including the above cast), and thus any successful analysis needs to model common patterns of polymorphism.

We have developed a static analysis that, augmented with 177 programmer annotations, proves the safety of 75% of the downcasts to structure types in Linux 2.6.17.1. The complete analysis consists of many components [11]; in this paper we focus on what we consider the most novel and difficult issue, the problem of analyzing *data polymorphism*, which is needed for 40% of the casts we are able to verify as well as most of the casts we are not able to verify. In Section 2 we expand the example, show why the cast is correct and motivate the problem of understanding data polymorphism. In Sections 3, 4 and 5 we reduce the problem of analyzing data polymorphism to discovering *structural relationships* and *structural correlations* and describe our algorithm. Section 6 presents experimental results, including examples that are beyond the reach of our fully automatic analysis, which we handle using programmer annotations. We postpone a discussion of our contributions until Section 3, after we have presented the extended example and the definition of structural relationships and structural correlations, the key ideas underlying our approach.

2. EXAMPLE

To verify the cast in `saa7146_buffer_timeout`, we must show the function's callers always pass a pointer to a value of type `saa7146_dmaqueue`. Now, `saa7146_buffer_timeout` is never called directly. In fact, it is mentioned in only two functions. The two cases are similar; one is shown below:

```
// drivers/media/common/saa7146_vbi.c
static void vbi_init(struct saa7146_dev *dev,
                     struct saa7146_vv *vv)
{
   INIT_LIST_HEAD(&vv->vbi_q.queue);
   init_timer(&vv->vbi_q.timeout);
   vv->vbi_q.timeout.function = saa7146_buffer_timeout;       (*)
   vv->vbi_q.timeout.data     = (unsigned long)(&vv->vbi_q);
   vv->vbi_q.dev              = dev;
   ...
}
```

The `timeout` field has type `timer_list`, a core kernel structure with a function pointer field `function` and an integer field `data`, among others. After the `function` field of the `timer_list` is assigned `saa7146_buffer_timeout`, where can the function eventually be called? The `function` field never has its address taken and is only assigned to a local variable `fn` in the core kernel function `__run_timers`:

*This work was supported in part by NSF grants CNS-050955 and CCF-0430378 with additional support from DARPA and gifts from Intel and IBM.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

```
// kernel/timer.c
static inline void __run_timers(tvec_base_t *base)
{
   struct timer_list *timer;
   ...
   while (...) {
      ...
      void (*fn)(unsigned long);
      unsigned long data;
      timer = list_entry(head->next,struct timer_list,entry);
      fn = timer->function;
      data = timer->data;
      ...
      fn(data);
      ...
   }
}
```

Function `__run_timers` repeatedly pulls `timer_list`s off of lists and calls `timer->function` with `timer->data`. This exposes the design intent of the `timer_list` structure: whatever is stored in the `function` field of a `timer_list` is called with the `data` field of that same `timer_list`. The `function` field points to `saa7146_buffer_timeout` for timers whose `data` field was written by `vbi_init`. After such writes, the `data` field points to the `vbi_q` field of a `saa7146_vv` structure, which has type `saa7146_dmaqueue`, which is just what `saa7146_buffer_timeout` expects. Thus, we know that if the function `saa7146_buffer_timeout` is only called with `vbi_q` for its `data` parameter, then the cast it performs is safe.

So, might `saa7146_buffer_timeout` be called with a parameter other than `vbi_q`? Two possibilities must be considered. First, the `data` field could be assigned another value (of a possibly different type) between the calls to `vbi_init` and `__run_timers`. Second, the `function` field could be called somewhere outside `__run_timers` with a parameter other than the `data` field of the `timer_list`. It turns out *both* possibilities actually occur (see below), but neither affects any `timer_list` containing `saa7146_buffer_timeout` in the `function` field.

Normally the `function` and `data` fields of a `timer_list` are written at the same time, shortly after the `timer_list` is created—after all, `__run_timers` requires both fields to be set. However, in a few places the `data` field is written without any corresponding write to the `function` field, such as in `tlclk_interrupt`:

```
// drivers/char/tlclk.c
static irqreturn_t tlclk_interrupt(...)
{
   ...
   if (int_events & HOLDOVER_01_MASK) {
      alarm_events->pll_holdover++;
      switchover_timer.expires = jiffies + msecs_to_jiffies(10);
      switchover_timer.data = inb(TLCLK_REG1);
      add_timer(&switchover_timer);
   }
   ...
}
```

Under certain circumstances `tlclk_interrupt` changes the `data` field of the global `switchover_timer` variable to a non-pointer integer value. But `switchover_timer` cannot alias a `timer_list` containing `saa7146_buffer_timeout` — `switchover_timer` is a statically allocated `timer_list`, while the timer manipulated by `vbi_init` is embedded in another structure.

The other way `saa7146_buffer_timeout` might receive a value other than `vbi_q` is if the `function` field is invoked with an argument other than the `data` field. There are five places in the kernel where the `function` field is invoked, and `__run_timers` is the only one where the `data` field is passed. The function `ctnetlink_del_conntrack` is representative of the other four:

```
// net/ipv4/netfilter/ip_conntrack_netlink.c
static int ctnetlink_del_conntrack(...)
{
   struct ip_conntrack *ct;
   ...
   ct = tuplehash_to_ctrack(h);
   ...
   if (del_timer(&ct->timeout))
      ct->timeout.function((unsigned long)ct);
   ip_conntrack_put(ct);
   return 0;
}
```

Instead of passing `ct->timeout.data` to `ct->timeout.function`, `ct` itself is passed, exploiting knowledge that `ct->timeout.data == ct` in this context. We know `ct->timeout` cannot alias the timer from `vbi_init`: the two timers are embedded in different types of structures. Thus, `ct->timeout.function` cannot invoke `saa7146_buffer_timeout`.

3. ANALYZING POLYMORPHISM

When `saa7146_buffer_timeout` is called by `__run_timers`, the data passed is provably generated by a previous call to `vbi_init`, not any of the hundreds of other assignments to the `data` field of a `timer_list` in the Linux kernel. The proof relies on two facts. First, the indirect call in `__run_timers` exploits a *structural relationship* between the function pointer target and function argument: these are fields of the same `timer_list` structure. In general, a structural relationship is a pair of locations reachable (via zero or more field accesses and dereferences) from a common base structure, or two locations reachable from the arguments to a common function. Second, the possible values of the locations in a structural relationship have *structural correlations* with one another: the `function` field of a `timer_list` is `saa7146_buffer_timeout` if and only if the `data` field was set by `vbi_init`, and similarly for the hundreds of other functions that may be used in a `timer_list`.

Structural relationships are related to the standard notion of type polymorphism in languages with more advanced type systems than C; commonly (but not exclusively) fields in a structural relationship would have related polymorphic types in statically typed functional or object-oriented languages. However, solving our problem requires more than identifying polymorphic fields, as we must also understand the actual contents of those fields, which means finding and correlating the field assignments. In the example the correlation is easily identified, as the structural correlation involves only two fields of a single structure and both fields are assigned in `vbi_init`. In more complex scenarios the structural relationship may span chains of dereferences across several structures, and the correlated assignments to the fields of the structural relationship may also be spread across multiple functions. It is the combination of identifying polymorphic data structures and the correlated side-effects to different parts of these structures in the heap that makes understanding data polymorphism a challenging problem.

Structural correlations are also related to standard notions in points-to analysis. Consider a field `a` that can take on values in set A and field `b` that can take on values in set B. Context-insensitive points-to analyses are often

too inaccurate in the sense that the cross-product $A \times B$ contains too many possibilities to be useful. By adding some form of context we split the fields `a` and `b` into multiple abstract locations `a1`, `a2`, ..., `b1`, `b2` ... representing smaller sets of runtime values, which will have analysis sets $A_1, A_2, \ldots, B_1, B_2, \ldots$ associated with them. The client of the points-to analysis must still consider the cross-products of values in these sets, but by adding context we hope that the set of pairs $\bigcup_{i,j} A_i \times B_j$ will be much smaller than the original cross product $A \times B$. The major difference with our approach is that structural correlation defines the desired output directly, without committing to a particular implementation strategy. Points-to analysis, in contrast, defines a particular framework in which the approach to improving precision is to refine (increase) the set of abstract locations.

No context-sensitive points-to methods have been shown to scale to programs the size of the Linux kernel, and based on the efforts that have tried [4], we believe it is necessary to take a different approach. The space consumption of a global points-to graph, particularly a context sensitive one, is difficult to control. Thus, our method does not build a global points-to graph. Instead, we first perform only local analysis of each function (which is in fact much more detailed than a points-to computation). At the interprocedural level we trade time for space, using an escape analysis to follow values through the program. This analysis queries the local analysis information, but does not build a global points-to graph, construct explicit contexts, or refine abstract locations. Because we do not build a global points-to graph or any global structures except for the structural correlations that are the output of the data polymorphism analysis, we do not encounter the memory consumption problems that appear to limit the scalability of context-sensitive points-to analyses.

Structural relationships and correlations are sufficiently general to tackle our polymorphism problem, the algorithm for which we break into two phases. First, we scan all indirect call sites to identify the important structural relationships holding between the function pointer used to invoke the call and the data (or other function pointers) reachable from the arguments to the call (Section 4). Second, we take in turn all the structural relationships identified for some indirect call site by the first phase, and for each of these identify all the possible structural correlations between particular functions and values which could exist for that relationship (Section 5).

Our algorithm is sound in the sense that if it identifies a structural relationship and associated structural correlations, it is guaranteed that *all* of the possible structural correlations for that structural relationship have been discovered; the algorithm has a complete view of the possible combinations of values that can be assigned to the fields involved in the structural relationship. Our algorithm is conservative in that it is not guaranteed to discover every structural relationship with non-trivial correlations, and even for the structural relationships it identifies as important it may fail to compute a set of structural correlations. Any field not in a structural relationship, or in a structural relationship that could not be successfully analyzed, is conservatively assumed to be able to take on any possible value for the field, independent of the values of other fields.

In summary, our main contributions are:

- We introduce structural relationships and structural correlations, which characterize the desired output of any analysis of data polymorphism without implying a particular implementation technique.
- We present an algorithm for computing structural correlations that is substantially different from conventional points-to analyses and has advantages for analyzing very large systems. In particular, we combine very precise but separate local analysis of individual functions with demand-driven and space-efficient interprocedural search algorithms.
- The core component of our search algorithm is an interprocedural escape analysis that may be of independent interest. The novel aspect is tunable precision, allowing us to conduct escape analysis at different granularities and use the most precise analysis which terminates with acceptable cost. Because both the cost and precision of an escape analysis query is unpredictable, the ability to try different strategies is very important.
- We give numerous examples of data polymorphism from the very simple to the very involved, including examples that our system cannot handle fully automatically. While the simple examples can be expressed in modern languages using parametric polymorphism, the most involved examples are not readily expressible in any static type system known to us, and furthermore, we are unaware of any previous literature where such coding patterns are described. These examples point out future challenges for both static analysis and static type systems in obtaining more expressive and automated systems for checking properties of large systems.
- We give the results of a large experiment in which we are able to statically verify 75% of downcasts to structure types in a version of the Linux kernel, out of a population of 28767.

Most of the components of our system have been described previously. It is the system architecture, the way the components are assembled, that is new. Thus, with a few exceptions for key aspects of our approach (structural relationships, correlations, and some aspects of the interprocedural escape analysis), we describe our approach at a relatively high level; details may be found in [11].

4. STRUCTURAL RELATIONSHIPS

A *trace* is an access path: a series of field accesses and pointer dereferences beginning with a global variable, local variable, function parameter, or allocation site. For example, `x->f.g->h` is a trace starting from variable `x`. A *relative trace* (or *rtrace*) drops the starting variable or allocation site; it is a pure sequence of field accesses and dereferences.

Structural relationships are recorded in two maps, one for structural relationships on types and one for functions:

$$\text{SR\_Type} \quad : \quad (callsite \times trace) \Rightarrow 2^{(type \times rtrace \times rtrace)}$$

$$\text{SR\_Func} \quad : \quad (callsite \times trace) \Rightarrow 2^{(trace \times trace)}$$

Consider a function `g` with an indirect call `f(x)` at call site `I`. Let `AT` be the trace which the call uses to access `x` (in

this case a function argument, and more generally a sequence of field selections/dereferences from an argument). Now $(\texttt{C}, \texttt{RFT}, \texttt{RT}) \in \text{SR\_Type}(\texttt{I}, \texttt{AT})$ if there is a structure of type C such that the function pointer f is at relative trace RFT from C and x is at relative trace RT from C. Also, $(\texttt{FT}, \texttt{T}) \in \text{SR\_Func}(\texttt{I}, \texttt{AT})$ if the function pointer f is reachable from an argument to g with trace FT and x is also reachable from one of g's arguments with trace T. For the indirect call in __run_timers and the call's first argument, a single structural relationship is found for SR_Type:

(timer_list, .function, .data)

Computing structural relations is a straightforward intraprocedural analysis. Taking a tuple (I,AT) as input, we determine the trace FT through which the indirect call is invoked, and the trace T passed as argument AT to the indirect call (the memory model accounts for all prior assignments and control flow in the function invoking I [19, 12]). Occasionally there may be multiple different values for FT or T, depending on the path taken to reach I (an example of this, __dentry_open, is shown in Section 5.3). In such cases we compute the relationships separately for each possible FT/T.

There is a problem, however. While using the plain arguments is sufficient for __run_timers, some calls are concerned not with an argument, but a field or transitive field of an argument (again, see Section 5.3). In general the amount of data reachable from each argument, and thus the number of possible structural relationships, is unbounded. To bound the number of structural relationships, we focus on relationships between function pointers and untyped data—void* pointers and integers which could be pointers in disguise (such as the argument to saa7146_buffer_timeout). These relationships are the most likely to have meaningful structural correlations, as well as being the most useful to the casting analysis. The argument traces AT we consider are:

- void* call arguments,
- void* fields of call arguments (or fields of fields, transitively, without following dereferences),
- any trace that might be cast by a target of the indirect call, as determined by a separate interprocedural analysis to determine possible function pointer targets and a prepass to look for casts in each function.

5. STRUCTURAL CORRELATIONS

For each structural relationship for some function call we represent the correlations that may hold:

$$\text{SC\_Type} \quad : \quad (name \times rtrace \times rtrace) \Rightarrow 2^{(name \times name \times trace)} + \top$$

$$\text{SC\_Func} \quad : \quad (name \times trace \times trace) \Rightarrow 2^{(name \times name \times trace)} + \top$$

Each $(\texttt{FNPTR}, \texttt{FN}, \texttt{DT}) \in \text{SC\_Type}(\texttt{C}, \texttt{FT}, \texttt{T})$ is a possible correlation for the (C,FT,T) structural relationship. For a value of type C, the value assigned to trace FT may be the function FNPTR, and the value assigned to trace T may be the value of DT when accessed from some call to FN; the function name FN is needed to give the context in which trace DT is interpreted. The relation SC_Func is similar. For the timer_list relationship (timer_list,.function,.data), there is a single correlation introduced by vbi_init:

(saa7146_buffer_timeout, vbi_init, &vv->vbi_q)

There are hundreds of correlations for this relationship, but no other uses saa7146_buffer_timeout for the function pointer.

To compute correlations for a structural relationship R, we first compute tuples (WFN,WFT,WT), where function trace WFT and trace WT may be set for R within function WFN. For SC_Type, WFN includes functions that write the fields of the relationship, and WFT and WT are the possible pairs of values written to those fields. If only one field is written, the other reflects the field's initial value. For SC_Func, WFN includes functions calling the function containing the indirect call I, and WFT and WT are the corresponding values passed at that call site in WFN.

The actual correlations (FNPTR,FN,DT) are computed from the triples (WFN,WFT,WT) by converting the trace WFT to one or more concrete function names FNPTR via any of the following methods:

- Use an escape analysis to determine where WFT came from and which functions it could refer to (Section 5.1).
- Follow transitive structural relationships between WFT and WT (Section 5.2). If WFT and WT are derived from the same structure, or both passed into the current function, they share a relationship whose correlations are a superset of the possible values for WFT and WT.
- If WFN is always invoked through an indirect call, look for structural relationships between WFT and the function pointer used to invoke WFN (Section 5.3).

Each of these approaches either fails or generates an overapproximation of the values of WFT and WT. If all approaches fail, we set SC_Type or SC_Func to $\top$; we could not capture the effect of all writes affecting the relationship. Otherwise, we take the intersection of all the result sets to get the tightest overapproximation we can for the correlations on (WFN,WFT,WT).

5.1 Escaped Correlations

We have developed an escape analysis determining where a value escaped from or where it may escape to. As mentioned in Section 3, we use escape analysis to avoid the unpredictable space consumption of a global points-to graph. The escape analysis' most novel aspect is tunable precision, which we discuss further below.

Our escape analysis is built upon a path-sensitive, intraprocedural memory and alias analysis that computes all aliases for each memory location accessed within a function body or loop in the manner of [19, 12]. Given a pair of traces, this per-function analysis returns the path-sensitive condition within that function under which the two traces alias. Note that if the condition is *false*, the traces cannot alias.

The escape analysis is demand driven, flow insensitive and has limited context sensitivity, but suffices for determining the functions referred to by many values of WFT. Consider a function trace WFN in function f. Escape_Backward(f,WFN), the set of functions that could flow to WFN, is calculated as follows (we do not describe Escape_Forward(,) which is symmetric). First, f's local information is queried to determine traces WFN is equivalent to, and in particular whether WFN is derived from a global or local variable, an argument of f, a field of a heap-allocated data structure, or a constant

(a concrete function name). The most interesting cases are handled as follows:

- If `WFN` aliases a function name, we return a singleton set containing that function.
- If `WFN` aliases an argument `x` of `f`, we return the union over all the following sets. Without loss of generality, assume `x` is the only argument of `f`.
  - For any direct call `f(e)` occurring in a function `g`, we compute `Escape_Backward(g,ET)`, where `ET` is the trace for `e`.
  - If the address of `f` is taken in function `h`, we also compute the set of indirect call sites of `f` via `Escape_Forward(h,FFN)`, where `FFN` is the trace of the location to which `f` is assigned. We add to the output `Escape_Backward(k,AT)` for each such indirect call in a function `k` with argument trace `AT`.
- If `WFN` aliases a structure field, then we must compute both forwards and backwards escape information for that field to see what assignments to the field may flow to `WFN`.

The output of the escape analysis is the set of concrete function names that can flow to `WFN`. In the simplest cases, such as in `vbi_init`, `WFT` is already a named function and the escape analysis gives us an exact singleton set. Now consider the function `vbi_init` without the write to `vv->vbi_q.timeout.function` (i.e., the line marked `(*)` is removed). In this case, the value of `WFT` is the value of the function pointer on entry to `vbi_init`, which is simply `*(vv->vbi_q.timeout.function)`. To compute the correlations, we need to know what values this function pointer can have. There are several ways to determine which concrete functions escape to this value: we can examine values assigned to the `.function` field of a `timer_list` anywhere in the program, values assigned to the field `.timeout.function` of a `saa7146_dmaqueue`, or to `.vbi_q.timeout.function` of a `saa7146_vv`, or values passed to `vbi_init` through the function argument `vv->vbi_q.timeout.function`. These are ordered by increasing precision: any value assigned to the `.timeout.function` field of a `saa7146_dmaqueue` is also assigned to the `.timeout` field of a `timer_list`, but not vice versa. A low level of precision may be too imprecise. However, the escape analysis cannot always determine the set of concrete functions for a value at a higher level of precision, because it may be too expensive to explore all the possibilities at a very fine level of granularity. In practice the best level of precision varies widely; we try several and use the most precise result that succeeds.

In this example, escape analysis using `.function` finds every function that could be assigned to *any* `timer_list`, a uselessly imprecise overapproximation. On the other hand, using `vv->vbi_q.timeout.function` follows `vv` everywhere it is passed in the code, and the escape analysis fails after exceeding a resource threshold. Escaping using `.timeout.function` or `.vbi_q.timeout.function` yields the correct result, finding the only value that is assigned directly to this field chain is `saa7146_buffer_timeout` and that the address of `.timeout` is not passed anywhere which will lead to the `function` field being written.

5.2 Transitive Correlations

Sometimes structural relationships are dependent on one another. Consider this code from the Linux IRQ subsystem:

```
// kernel/irq/manage.c
int request_irq(unsigned int irq,
       irqreturn_t (*handler)(int, void *, struct pt_regs *),
       unsigned long irqflags, const char * devname, void *dev_id)
{
   struct irqaction * action;
   ...
   action = kmalloc(sizeof(struct irqaction), GFP_ATOMIC);
   if (!action) return -ENOMEM;
   action->handler = handler;   (*)
   action->flags = irqflags;
   cpus_clear(action->mask);
   action->name = devname;
   action->next = NULL;
   action->dev_id = dev_id;     (*)
   ...
}
```

Type `irqaction` has a function pointer `handler` called when a specific interrupt is received. The `handler` field is passed, among other things, the `void*` field `dev_id`, so `.handler` and `.dev_id` have a structural relationship R. Each `irqaction` is created within `request_irq`; note the function arguments `handler` and `dev_id`. There is thus another structural relationship R' between variables `handler` and `dev_id`, and because of the assignments marked `(*)` any correlations in R' are also correlations of R. We detect such dependencies between structural relationships in a manner similar to the handling of indirect call sites (Section 4). When computing structural correlations for the `handler` and `dev_id` fields of `irqaction`, we notice the assignments to those fields in `request_irq` participate in R' and add all correlations for R' to R. Because there may be cycles in the graph of dependencies between structural relationships, this process is iterated to a fixed point (i.e., until no new transitive correlations are discovered).

5.3 Dominating Indirect Calls

More complicated types of data polymorphism correlate data with multiple function pointers. These function pointers often manage the data's contents, and we can recover correlations from calls to these functions. Consider this function in the saa7146 driver:

```
// drivers/media/common/saa7146_fops.c
static ssize_t fops_read(struct file *file,
              char __user *data, size_t count, loff_t *ppos)
{
   struct saa7146_fh *fh = file->private_data;
   switch (fh->type) {
      ...
   }
}
```

The `file->private_data` pointer has type `void*`, and thus `fops_read` performs a cast we are interested in checking for type safety. What's going on with this function?

In keeping with Unix practice, user applications in Linux interact with many devices as if they were regular files. Linux has a common interface for defining new files: the `file_operations` structure, a table of 27 function pointers (though not all are used by each driver or filesystem).

```
// include/linux/fs.h
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file*, loff_t, int);
```

```
   ssize_t (*read) (struct file*, char __user*, size_t, loff_t*);
   ssize_t (*aio_read) (...);
   ssize_t (*write) (...);
   ssize_t (*aio_write) (...);
   ...
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *);
   int (*release) (struct inode *, struct file *);
   ...
};
```

Interaction with a `file` is primarily through the function pointers in the `f_op` field pointing to the file's `file_operations`. For example, `vfs_read` reads out of a file:

```
// include/linux/fs.h
struct file {
   ...
   struct dentry                *f_dentry;
   struct vfsmount              *f_vfsmnt;
   const struct file_operations  *f_op;
   ...
   void               *private_data;
   ...
};

// fs/read_write.c
ssize_t vfs_read(struct file *file,
                 char __user *buf, size_t count, loff_t *pos)
{
   ...
   ret = security_file_permission (file, MAY_READ);
   if (!ret) {
      if (file->f_op->read)
         ret = file->f_op->read(file, buf, count, pos);
      ...
   }
   return ret;
}
```

The system call `sys_read` invokes `vfs_read` directly. To allow `vfs_read` and other top-level file operations to interact with `saa7146` devices, the `saa7146` driver creates a `file_operations` structure whose `read` field is set to `fops_read`. Note `vfs_read` will indirectly call `fops_read`.

```
// drivers/media/common/saa7146_fops.c
static struct file_operations video_fops =
{
   .owner     = THIS_MODULE,
   .open      = fops_open,
   .release   = fops_release,
   .read      = fops_read,
   .write     = fops_write,
   ...
};
```

Now that `fops_read` can be invoked, why is the cast it performs correct? Interestingly, `video_fops` is never directly assigned to any `file->f_op`, and writes to the `f_op` field are never directly correlated with writes to the `private_data` field. A more elaborate mechanism is in use. Function `__dentry_open`, which opens a file, sets the `f_op` field and calls its `open` method.

```
// fs/open.c
static struct file *
__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
              int flags, struct file *f,
              int (*open)(struct inode *, struct file *))
{
   ...
   f->f_dentry = dentry;
   f->f_vfsmnt = mnt;
   f->f_op = fops_get(inode->i_fop);
   ...
   if (!open && f->f_op)
      open = f->f_op->open;
   if (open) {
      error = open(inode, f);
      if (error)
         goto cleanup_all;
   }
   ...
}
```

Returning to the `video_fops` used to store `fops_read`, we see the corresponding open function is `fops_open`, which sets the `private_data` field of the file to the value expected by `fops_read`.

```
// drivers/media/common/saa7146_fops.c
static int fops_open(struct inode *inode, struct file *file)
{
   struct saa7146_fh *fh = NULL;
   ...
   fh = kzalloc(sizeof(*fh),GFP_KERNEL);
   if (NULL == fh) {
      ...
      goto out;
   }
   file->private_data = fh;
   fh->dev = dev;
   fh->type = type;
   ...
}
```

For the polymorphic data analysis we need to correlate the `f_op->read` field of a `file` with the `private_data`. We cannot do this by looking for matched writes of `f_op` and `private_data`, but instead by matching up the `f_op->open` function and the writes it performs with the `f_op->read` function.

The write we are most concerned with is in `fops_open` to `file->private_data`. We are interested in the possible values for `file->f_op->read` here, and while that value is not written in `fops_open` we can get information about it from the call stack. Now, `fops_open` is only called indirectly through `__dentry_open` and a few similar functions. In each such function we can prove `fops_open` is only called through `file->f_op->open`: the code is some variant of `file->f_op->open(inode, file)`. We thus know in `fops_open`, `file->f_op->open == fops_open`. When this equality holds, what are the possible values for `file->f_op->read`? If we track the structural relationship for type `file_operations` between its `open` and `read` fields, we can answer this question with the resulting correlations.

Finding the correlations for this `file_operations` relationship is straightforward, as the `open` and `read` fields are always written in synchronization with each other, almost always in a global initializer. With `fops_open` in the `open` field, the only value for the `read` field is `fops_read`, which is thus correlated with the value written to `private_data` in `fops_open`.

This dominating-caller technique is geared towards relationships involving function pointer tables with an `open`-type method that fills in private data for other methods in the table to access. The technique in whole is:

1. For a function `FN`, find a function pointer trace `XFT` such that `FN` is called only when `XFT` is a particular function `XFNPTR`. This dominance relation holds for `FN` if either:

   - `FN` is only called indirectly and `XFT` is the invoked function pointer at each parent call site. In this case `XFNPTR` = `FN`.

- Each PFN that can invoke FN is itself dominated by calls where XFNPTR = YFT for some YFT in PFN.

Searching for dominators is k-limited to avoid unbounded call graph exploration; using $k = 5$ has been sufficient.

2. Look for a structural relationship on a struct type between XFT and WFT. Normally the type is a function pointer table like file_operations.

3. If there is such a relationship, then for each correlation between XFNPTR and some ZFT, the possible values for WFT are the union over the the ZFT. Normally each ZFT is a particular function; if not, resolve with the escape analysis as in Section 5.1.

6. RESULTS

The version of Linux we analyzed, 2.6.17.1, contains about 4.4 million lines of code and 11976 indirect call sites. Of these call sites, 7850 (66%) involve structural relationships—a parameter to the call was either a void* pointer or was a structure containing a void* field. From these call sites, 8830 structural relationships were identified, and an additional 5939 relationships were added by transitive correlations (see Section 5.2), for a total of 14769 structural relationships. Of these, 9601 were between structure fields, and 5168 between the arguments to a function.

We successfully found the correlations for 10416 (71%) relationships, including 5750 (60%) structure field relationships and 4666 (90%) function argument relationships (the remainder were marked as failed). Of the 7850 call sites with relationships, correlations were found for at least one relationship at 6883 (88%) sites.

Our parallel implementation of the analysis took 3 hours and 42 minutes to run on a 50 core cluster, using 130 hours of CPU time (the analysis was written using a logic programming language [11], which in our experience is much easier to develop analyses in than C, but incurs a 20x to 40x slowdown over C). Analysis timed out on 522 functions, or .5% of all functions analyzed; these timeouts can cause us to unsoundly underapproximate the correlations. We examined many of these timeouts, which were generally caused by functions where the analysis would have ultimately failed anyway and thus did not affect the generated correlations.

The results of this analysis are crucial for our broader analysis for proving the safety of type casts [11]. Out of a population of 28767 downcasts, we prove the safety of 21637, 75.2% of the total. Of the proved casts, 8754 or 40% require the polymorphic relationships identified here.

A small group of structures with polymorphic relationships are responsible for most proved casts: 173 different structures have some associated relationship used to prove at least one cast. Of the 8754 casts proved using polymorphism, 7521 (86%) use relationships from a set of 26 structures used to prove 50 or more casts each, and 6000 (69%) use relationships from a set of 10 structures used to prove 200 or more casts each. This latter set includes both the file structure (used to prove 736 casts) and timer_list structure (used to prove 408 casts).

Analyzing polymorphic relationships with sufficient precision for the casting analysis required 177 annotations (for several million lines of code). These annotations are trusted: they are assumed by the analysis and must be checked manually. Annotations are needed for three broad reasons:

- General analysis imprecision leading to results too imprecise for the casting analysis. This imprecision accounts for about 75% of the annotations we needed.

- The initialization of a structure may not fit the inference techniques our analysis uses to find correlations. The fit is often close, and we can use annotations to adjust the inference to match the initialization.

- The structure's polymorphism might not fit the model of structural relationships our analysis uses. We can sometimes fit these cases so that we can capture the needed correlations, even if our analysis of the structure's internals is largely incomplete.

The following subsections give examples of each category.

6.1 Analysis Overapproximation

When initialization of multiple fields of a data structure is split across many functions, we need precise knowledge of which fields are uninitialized, NULL, or non-NULL at various control points to generate precise correlations. Consider again the __dentry_open function from Section 5.3. In this example, we are interested in structural relationships between the private_data field of a file and the read and other fields of that file's f_op table. The f_op is written in __dentry_open, and private_data is written in the indirect call to open.

Our analysis sees the f_op write in __dentry_open and no private_data write, and so correlates all possible values of f_op->read (all file read functions in existence) with the input value f->private_data. Simply inlining the possible targets of open cannot help; some open methods do not set private_data, as those filesystems never use that field. Now, __dentry_open is only called during initialization of f, and the only possible value for f->private_data at entry to __dentry_open is NULL. Unfortunately, our system misses that f->private_data is NULL due to tricky initialization code. Usually __dentry_open is called through dentry_open, which directly allocates a file with NULL contents through get_empty_filp; this case is easy to analyze.

```
// fs/open.c
struct file *dentry_open(struct dentry *dentry,
                         struct vfsmount *mnt, int flags)
{
   struct file *f;
   ...
   f = get_empty_filp();
   if (f == NULL) {  ...   }
   return __dentry_open(dentry, mnt, flags, f, NULL);
}
```

The difficult case is lookup_instantiate_filp, another caller of __dentry_open, which passes in as the file argument nd->intent.open.file, a pointer to data allocated by its own callers.

```
// fs/open.c
struct file *lookup_instantiate_filp(struct nameidata *nd,
               struct dentry *dentry,
               int (*open)(struct inode *, struct file *))
{
 ...
 nd->intent.open.file = __dentry_open(dget(dentry), mntget(nd->mnt),
        nd->intent.open.flags - 1, nd->intent.open.file, open);
 ...
}
```

While the nd->intent.open.file pointer is always either NULL or points to an empty file in this function, it is allocated several levels up the call chain and across potentially multiple indirect calls. We use one annotation to disable correlations between fields of the file structure within __dentry_open.

In general, the annotations we used to fix imprecision either disable a portion of the analysis for some function (where doing so will not cause the correlations to be underapproximated) or correct some intermediate analysis information to increase the precision of the correlations.

6.2 Unhandled Initialization

Some data structures have important structural relationships but the initialization is a poor fit for our inference algorithm. For example, in some sound PCM layer structures snd_pcm_ops is a function pointer table used by snd_pcm and its children:

```
// include/sound/pcm.h

struct snd_pcm {
   struct snd_card *card;
   ...
   struct snd_pcm_str streams[2];
   ...
   void *private_data;
   void (*private_free) (struct snd_pcm *pcm);
   ...
};

struct snd_pcm_str {
   int stream;
   struct snd_pcm *pcm;
   unsigned int substream_count;
   unsigned int substream_opened;
   struct snd_pcm_substream *substream;
   ...
};

struct snd_pcm_substream {
   struct snd_pcm *pcm;
   struct snd_pcm_str *pstr;
   void *private_data;
   ...
   struct snd_pcm_ops *ops;
   ...
   struct snd_pcm_substream *next;
   ...
};
```

Each snd_pcm has two child snd_pcm_str structures in its streams field; each snd_pcm_str has a list substream of snd_pcm_substream structures that are linked through the next field. Each object has pointers back to its parents.

There are important structural relationships between the function pointers in the ops field of a snd_pcm_substream and its private_data field. Writes to the ops and private_data fields of snd_pcm_substream are not correlated in the usual way. Instead of writing both fields together, functions initializing the parent snd_pcm write to the ops field of *all* the associated substreams with the snd_pcm_set_ops helper function, but only write to the private_data of the parent snd_pcm. An example is in snd_atiixp_pcm_new, which is called during device probe and allocates and initializes a new snd_pcm.

```
// sound/core/pcm_lib.c
void snd_pcm_set_ops(struct snd_pcm *pcm, int direction,
                     struct snd_pcm_ops *ops)
{
   struct snd_pcm_str *stream = &pcm->streams[direction];
   struct snd_pcm_substream *substream;
   for (substream = stream->substream; substream != NULL;
        substream = substream->next)
      substream->ops = ops;
}

// sound/pci/atiixp_modem.c
static int __devinit snd_atiixp_pcm_new(struct atiixp_modem *chip)
{
   struct snd_pcm *pcm;
   int err;
   ...
   err = snd_pcm_new(chip->card, ..., &pcm);
   if (err < 0) return err;
   snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                   &snd_atiixp_playback_ops);
   snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                   &snd_atiixp_capture_ops);
   pcm->dev_class = SNDRV_PCM_CLASS_MODEM;
   pcm->private_data = chip;
   ...
}
```

After initialization, the substream's ops is set but not its private_data. This state persists until the substream is opened in snd_pcm_open_substream, which looks up the substream via snd_pcm_attach_substream, which scans the substreams in the snd_pcm, finds one that is not in use, and sets its private_data to the private_data of the parent snd_pcm.

```
// sound/core/pcm_native.c
int snd_pcm_open_substream(struct snd_pcm *pcm, int stream,
                           struct file *file,
                           struct snd_pcm_substream **rsubstream)
{
   struct snd_pcm_substream *substream;
   int err;
   err = snd_pcm_attach_substream(pcm, stream, file, &substream);
   if (err < 0) return err;
   ...
   if ((err = substream->ops->open(substream)) < 0) goto error;
   ...
}

// sound/core/pcm.c
int snd_pcm_attach_substream(struct snd_pcm *pcm, int stream,
                     struct file *file,
                     struct snd_pcm_substream **rsubstream)
{
   struct snd_pcm_str * pstr;
   struct snd_pcm_substream *substream;
   ...
   pstr = &pcm->streams[stream];
   if (pstr->substream == NULL || pstr->substream_count == 0)
      return -ENODEV;
   ...
   for (substream = pstr->substream; substream;
        substream = substream->next)
      if (!SUBSTREAM_BUSY(substream)) break;
   if (substream == NULL) return -EAGAIN;
   ...
   substream->private_data = pcm->private_data;
   substream->ffile = file;
   pstr->substream_opened++;
   *rsubstream = substream;
   return 0;
}
```

By correlating a write to snd_pcm->private_data with calls to snd_pcm_set_ops, a PCM driver ensures when the substream is opened the correlation between private_data and the ops used in snd_pcm_set_ops is introduced as a correlation in the snd_pcm_substream structural relationship. Our annotations add correlations for snd_pcm_substream when snd_pcm->private_data is written or snd_pcm_set_ops is called, not when the ops or private_data fields of the snd_pcm_substream itself are written.

6.3 Unhandled Polymorphism

The most interesting uses of polymorphism are those our analysis cannot even express. There are not many of these, but they are generally important. We have annotated one such case, the Sysfs filesystem, providing a mechanism to check the casts performed by clients of Sysfs with the usual limitation that the annotations are trusted; we assume Sysfs follows the annotated behavior. In this section we describe the interface Sysfs uses to expose its polymorphism, which we have annotated, and the internal invariants Sysfs maintains for this interface, which our analysis has little understanding of.

Sysfs provides a mechanism for userspace programs to query and update attributes of the drivers and associated devices by accessing files in the `/sys` directory. To the driver writer, this functionality is behind a simple polymorphic interface, which relates a kernel object `kobj` (each device used in Sysfs has its own kernel object) with an attribute with a name and access mode (read, read/write, etc.).

```
// include/linux/sysfs.h
int sysfs_create_file(struct kobject * kobj,
                      const struct attribute * attr);

struct attribute {
   const char      * name;
   struct module   * owner;
   mode_t          mode;
};
```

The driver uses `sysfs_create_file` by passing the device's kernel object and the attribute to associate with the device.

```
// drivers/block/aoe/aoeblk.c
static ssize_t aoedisk_show_state(struct gendisk * disk, char *page)
{
   struct aoedev *d = disk->private_data;
   return snprintf(page, PAGE_SIZE, ...);
}

static struct disk_attribute disk_attr_state = {
   .attr = {.name = "state", .mode = S_IRUGO },
   .show = aoedisk_show_state
};

static void aoedisk_add_sysfs(struct aoedev *d)
{
   sysfs_create_file(&d->gd->kobj, &disk_attr_state.attr);
   sysfs_create_file(&d->gd->kobj, &disk_attr_mac.attr);
   sysfs_create_file(&d->gd->kobj, &disk_attr_netif.attr);
   sysfs_create_file(&d->gd->kobj, &disk_attr_fwver.attr);
}
```

In this example, there is a correlation where the `disk` parameter to `aoedisk_show_state` is equal to the `d->gd` value as passed into a call to `aoedisk_add_sysfs`. We need to know this correlation to show that the cast of `disk->private_data` performed by `aoedisk_add_sysfs` is correct. We use a total of 17 annotations to capture the correlations introduced by calls to `sysfs_create_file` and several wrappers which create Sysfs files for particular kinds of devices.

These annotations do not address the internal invariants of Sysfs, the machinery hidden behind `sysfs_create_file` and the filesystem itself which ensures `aoedisk_show_state` is called with the right value. The remainder of this section describes these invariants. Calling `sysfs_create_file` eventually leads to a file with the following file operations (see Section 5.3 for a description of `file_operations`).

```
// fs/sysfs/file.c
const struct file_operations sysfs_file_operations = {
   .read      = sysfs_read_file,
   .write     = sysfs_write_file,
   .llseek    = generic_file_llseek,
   .open      = sysfs_open_file,
   .release   = sysfs_release,
   .poll      = sysfs_poll,
};
```

When a user tries to read this file, the `sysfs_read_file` function is called, which invokes `aoedisk_show_state` on the correct `disk` argument to get the state of the disk.

```
// fs/sysfs/file.c
static ssize_t sysfs_read_file(struct file *file, char __user *buf,
                               size_t count, loff_t *ppos)
{
   struct sysfs_buffer * buffer = file->private_data;
   ...
   if (buffer->needs_read_fill) {
      fill_read_buffer(file->f_dentry,buffer);
   }
   ...
}

static int fill_read_buffer(struct dentry * dentry,
                            struct sysfs_buffer * buffer)
{
   struct attribute * attr = to_attr(dentry);
   struct kobject * kobj = to_kobj(dentry->d_parent);
   struct sysfs_ops * ops = buffer->ops;
   ...
   count = ops->show(kobj,attr,buffer->page);
   ...
}

// fs/sysfs/sysfs.h
static inline struct attribute * to_attr(struct dentry * dentry)
{
   struct sysfs_dirent * sd = dentry->d_fsdata;
   return ((struct attribute *) sd->s_element);
}

static inline struct kobject * to_kobj(struct dentry * dentry)
{
   struct sysfs_dirent * sd = dentry->d_fsdata;
   return ((struct kobject *) sd->s_element);
}

// block/genhd.c
#define to_disk(obj) container_of(obj,struct gendisk,kobj)

static ssize_t disk_attr_show(struct kobject *kobj,
                              struct attribute *attr, char *page)
{
   struct gendisk *disk = to_disk(kobj);
   struct disk_attribute *disk_attr =
      container_of(attr,struct disk_attribute,attr);
   if (disk_attr->show) disk_attr->show(disk,page);
}

static struct sysfs_ops disk_sysfs_ops = {
   .show   = &disk_attr_show,
   .store  = &disk_attr_store,
};
```

Now the function `sysfs_read_file` calls the helper function `fill_read_buffer`, which gets an `attribute` and `kobject` from the file and performs an indirect call `ops->show` to fill in the data from the attribute which will be returned by the file read. If the attribute read is `disk_attr_state.attr` (or any other attribute of a `gendisk`), the `ops` points to `disk_sysfs_ops`, and `ops->show` calls `disk_attr_show`, which backs out the kernel object pointer to the containing `gendisk` (`d->gd` in the call to `aoedisk_add_sysfs`) and the attribute pointer to the containing `disk_attribute` (`disk_attr_state`). Finally, `disk_attr_state.show` points to `aoedisk_show_state`, completing the call chain from `sysfs_read_file`.

This example assumes numerous data invariants which must hold or else the indirect calls will break. Our analysis can capture some of these invariants, but user annotations are required for the rest; for more details, see [11]. More complete automatic checking for these properties at this scale is well beyond what is currently feasible with existing techniques.

7. RELATED WORK

Our analysis can be characterized as simultaneously scaling to large programs (millions of lines of code), being a verifier (i.e., proving properties, in contrast to finding bugs), and being highly *heap sensitive*, meaning simply that to be successful it requires a relatively deep understanding of the relationships between data structures in the heap. Several bug-finding (non-verifying) efforts have scaled to systems of the size we consider; representative examples include [13, 19, 3]. Fewer verifiers have been demonstrated to work on million line programs and these have focused on finite-state properties; these systems are subject to the caveat (as is our system) that portions of the analysis may be unsound due to time-outs and other resource limits for a small portion of the analysis [8, 14, 4]. We are not aware of any previous work on verifying type casts that scales to programs of the size that we analyze, and more generally we are not aware of any verification system that is heap sensitive on multi-million line programs.

C and C++ are alone among widely used typed languages today in not providing type safety guarantees. Consequently, research has sought to ensure that C programs are type safe, or to replace C with similarly expressive type safe alternatives. Most work focuses not just on type safety, but memory safety as well (ensuring `NULL` or dangling pointers are not dereferenced, buffers do not overrun, and so forth).

Siff et. al. [18] describe rules for physical subtyping in C and examine the casts in several hundred thousand lines of code. They find that about 85% of the downcasts involving structure types in C are between `void*` or `char*` and a structure, rather than between different structure types. In the Linux kernel version we analyzed we found far fewer casts involving structure subtyping — just 459 out of 44910 casts, or 1%, and involving just 44 different supertypes. For these casts we use the same physical subtyping rules as [18] to determine compatibility between the structures. However, rather than just counting the number of downcasts in a program our interest is in proving these casts correct.

Loginov et. al. [16] compute type information for C programs at runtime and check the program's behavior against these types to find type safety violations. Since virtually any access in C might be type unsafe, virtually all memory accesses are instrumented by this method, leading to an average slowdown of greater than 20 times the original program's runtime.

HAVOC [15] is a static analysis system for C programs that uses function preconditions, postconditions, and loop invariants to perform modular verification of memory safety and other properties. HAVOC has recently been used to verify type safety for a few Windows device drivers [5]. HAVOC provides far stronger guarantees about a program than the casting analysis we present; we are only checking downcasts to structure types, while HAVOC checks these as well as downcasts to other types, use of the `container_of` macro to jump to a structure's base pointer, buffer overflows, and all other ways type safety might be violated. However, to completely verify 5000 lines of code, HAVOC requires 35 changes to the code, 36 trusted annotations (annotations which, like our annotations, are not checked for correctness), and 153 untrusted annotations (which are checked for correctness). At these rates, annotating and checking a system the size of the Linux kernel would require several hundred thousand lines of annotations.

CCured [17, 7] uses pointer type qualifiers in combination with runtime checks to check type and memory safety in C with fairly low overhead. Pointers used in downcasts are transformed into 'fat' `WILD` pointers, structures which contain both the pointer and additional bounds and runtime type information to perform the appropriate checks at accesses to the pointer. The initial version of CCured [17] would mark as `WILD` any pointer whose value might have been used in a downcast or might in the future be downcast (according to a global flow- and context-insensitive algorithm). For polymorphic structures such as `file` and `timer_list`, this would encompass all uses of the data which at any point were stored in their `void*` data fields. An improvement [7] allows most pointers which are downcast to be less than fully `WILD` at the cost of limited runtime type information attached for checking the downcast is safe. After the downcast and checks are performed, the result is a `SAFE` pointer which can be accessed in the future with few additional checks.

Deputy [6] is a type system for C that uses a more lightweight approach than CCured, inserting runtime assertions where necessary without changing the in-memory layout of pointers and other structures. When dealing with downcasts from one type to another, Deputy soundly checks the cast at compile time provided the pointers are annotated with correct dependent types. The dependent types used by Deputy cover the parametric polymorphism as used in many of the Linux kernel data structures [2], but not other, rarer constructs such as pointers whose type depends on a program condition. Moreover, even if suitable polymorphic types are assigned for the various polymorphic structures in Linux, it is not clear that the Deputy checker can deal with many of the intricacies found in initialization of these structures; for example, the `f_op` field of a `file` may be freely changed so long as its `private_data` is `NULL` (Section 6.1).

Cyclone [10] is a C-like language that ensures memory and type safety, sharing many of the same features as CCured and Deputy. Pointers used in arithmetic can be either fat as in CCured, or be associated with a specific length as in Deputy. Casts are allowed in Cyclone, but only from a subtype to a supertype [1]; downcasts are disallowed. Types in Cyclone can be polymorphic [9] in a similar fashion to Deputy, again handling many of the polymorphic structures we have seen in Linux and removing the need for many downcasts. Still, Cyclone requires that the type over which a polymorphic structure is instantiated be set at the creation point of the structure, which breaks on initializers such as the `file` open example (Section 6.1).

Our approach to modeling polymorphic structures is more indirect than the approaches used by Deputy and Cyclone, and does not try to associate type variables with the structure declarations and concrete type instantiations at each point the structure is used. This allows us to handle cases such as the `file` open example, as we do not have to fix a type to a `file` at the points where it in fact has no type.

Finally, more modern languages than C, such as C++ and Java, have richer type systems that can directly express polymorphic interfaces a C programmer must construct by hand. For example, `timer_list` could be implemented as a C++ template structure, and `file_operations` could be implemented as virtual methods in a C++ class. Whether such languages are appropriate for a full-fledged operating system is a divisive topic; we observe, though, that the `file` open example illustrates the flexibility of C to write code which falls outside the usual approach of a C++ or Java program.

8. CONCLUSION

Big software systems are tremendously complex with all their details taken together. By focusing on downcasts we are able to peel away and characterize a small portion of this complexity. The combination of polymorphic data structures and initialization via assignment leads to important and sometimes complex relationships that are critical to proving basic safety properties of large systems. Understanding these heap invariants is a challenging static analysis problem, and we have shown that it can be solved automatically for many, but not all, of the common idioms in the Linux kernel. We suspect the results apply beyond Linux and C; in particular, we expect large systems written in untyped scripting languages will display similar phenomena, and we even suppose that similar implicit structural correlations can be found in large systems written in strongly typed languages such as Java, at least for properties of fields that are not directly enforced by the strong type system.

9. REFERENCES

[1] *Cyclone: User Manual*. http://cyclone.thelanguage.org/wiki/User Manual.

[2] *Deputy Manual*. http://deputy.cs.berkeley.edu/manual.html.

[3] Alex Aiken, Suhabe Bugrara, Iisil Dillig, Thomas Dillig, Peter Hawkins, and Brian Hackett. An overview of the saturn project. In *Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[4] Suhabe Bugrara and Alex Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, pages 325–338, 2008.

[5] Jeremy Condit, Brian Hackett, Shuvendu Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages*, 2009.

[6] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

[7] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Programming Language Design and Implementation*, 2003.

[8] Manuvir Das, Sorin Lerner, and Mark Seigle. Program verification in polynomial time. In *Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[9] Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28, 2006.

[10] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: a type-safe dialect of C. In *C/C++ Users Journal*, volume 23, 2005.

[11] Brian Hackett. *Type Safety in the Linux Kernel*. PhD thesis, Stanford University, 2010.

[12] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *Foundations of Software Engineering*, 2006.

[13] Seth Hallen, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[14] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

[15] Shuvendu Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages*, 2008.

[16] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *In Proceedings of FASE 2001: Fundamental Approaches to Software Engineering*, pages 217–232. Springer, 2001.

[17] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. In *Principles of Programming Languages*, 2002.

[18] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in C. In *Foundations of Software Engineering*, 1999.

[19] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Principles of Programming Languages*, 2005.