TYPE SAFETY IN THE LINUX KERNEL

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF
COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Brian Hackett
April 2011

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Monica Lam)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dawson Engler)

Approved for the University Committee on Graduate Studies.

Abstract

Type casts are ubiquitous in Linux and other systems software. Casts reinterpret a pointer to a heap object of one type as a pointer to another type, and are the main way by which programmers can violate type safety, the notion that objects are used as a consistent type throughout their lifetime. A program which is not type safe can exhibit corruption and crashes. Moreover, the C language provides no guarantees about type safety, so responsibility for ensuring type safety falls entirely to programmers.

This thesis describes an approach to proving that type casts preserve type safety, specifically in the Linux kernel. This approach uses automated static analysis to infer the abstractions underlying each type cast: type casts are performed for a reason, often to implement polymorphism or another advanced programming language feature, and by identifying this reason the cast can be proved safe. We prove this safety for 75.2% of downcasts to structure types in Linux, out of a population of 28767.

Analyzing type casts in Linux requires deep reasoning about the heap at a scale far beyond the previous state of the art. This thesis describes the numerous analysis techniques we developed for this problem, which are broadly applicable to doing sound and precise analysis of the heap at a large scale.

Contents

List of Tables

List of Figures

Chapter 1

Overview

1.1 Introduction

Type casts are ubiquitous in Linux and other systems software. Casts reinterpret a pointer to one type as a pointer to another type, and are the main way by which programmers can violate *type safety*, the notion that objects are used as a consistent type throughout their lifetime. The C language provides no guarantees about type safety, so responsibility for ensuring type safety falls entirely to the programmers.

The focus of this thesis is on proving that type casts preserve type safety, specifically in the Linux kernel. Our approach is to use static analysis to infer the abstractions underlying a type cast — casts are performed for a reason, often to implement polymorphism or another advanced programming language feature, and by identifying this reason we can prove the cast is safe. We can prove this safety for 75.2% of downcasts to structure types in Linux, out of a population of 28767.

A program which is not type safe can exhibit corruption and crashes.

More fundamentally, such a program will be extremely difficult to understand. Suppose we are reading a program and see a read of the form `x->f`. A natural question is where that value came from. Which writes in the program could have supplied that value?

If somewhere else in the program we see a write to `y->f`, we observe that this write and the read could be to the same heap location, provided it is possible that `x == y`. What if the write is instead to a `y->g`, where the type of `y` is unrelated to that of `x`? In general, the write and read could be taken to be to the same heap location.

While aliasing between such terms as `x->f` and `y->g` **does** occur occasionally in most systems software, in the vast majority of cases it does not. Code which does not permit such aliasing is *type safe*, which we'll define as follows:

- An *access* to a field occurs when that field is either directly read or written (`x.f`, `x->f`), or has its address taken (`&x.f`, `&x->f`) and that pointer or a copy is later dereferenced.
- Any heap location (region of memory) accessed as a particular field is only ever accessed as that same field.

A violation of type safety can come about through several means:

- Using pointer arithmetic to skip around the fields of a structure, by taking either the base address or address of a structure's internal field, and incrementing or decrementing it to refer to a different field of that structure. However, it is fine to use the Linux `container_of` macro or a similar operation to obtain a structure's base address from the address of one of its fields.
- Misuse of the `container_of` macro, specifying the wrong base type or field for an internal field address, which will produce a

malformed pointer potentially referencing any field of the containing structure or even unallocated space.

- Using multiple fields of a `union` structure.
- Casting the same pointer into multiple incompatible types, allowing the data referenced to be accessed as different fields of the different types.

We target the last class, the problem of proving the correctness of casts, and even then only for casts to structure pointers. We focus on the Linux kernel, including all associated device drivers and file systems. The version we analyzed, 2.6.17.1, contains several million lines of code. How can we check that pointers to each structured value are always cast to a consistent type?

1.2 Example

Take a typical function in Linux, `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;

   ...
}
```

The first thing this function does is cast its integer parameter `data` to a pointer to a value of type `saa7146_dmaqueue`, and then proceeds

to access the contents of that structure. If `data` really is an integer and not a pointer then these accesses will crash the system, and if `data` is a pointer to an object that is not of type `saa7146_dmaqueue` then these accesses will lead to corruption and/or a crash. We can prove this cast is safe to perform, but doing so requires fairly deep reasoning about the heap, control flow and data flow of the system.

We need to know who calls `saa7146_buffer_timeout` and make sure they always pass in a pointer to a value of type `saa7146_dmaqueue`. The function `saa7146_buffer_timeout` is never called directly. In fact, it is only mentioned in two places, functions `vbi_init` and `video_init` within the saa7146 driver.

```
// 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;

   ...
}

// drivers/media/common/saa7146_video.c
static void video_init(struct saa7146_dev *dev,
                       struct saa7146_vv *vv)
{
   INIT_LIST_HEAD(&vv->video_q.queue);

   init_timer(&vv->video_q.timeout);
   vv->video_q.timeout.function = saa7146_buffer_timeout;
   vv->video_q.timeout.data     = (unsigned long)(&vv->video_q);
   vv->video_q.dev              = dev;
```

```
   ...
}
```

The `timeout` field used in these two functions is a value of type `timer_list`, a core kernel structure with the following signature:

```
// include/linux/timer.h
struct timer_list {
   struct list_head entry;
   unsigned long expires;

   void (*function)(unsigned long);
   unsigned long data;

   struct tvec_t_base_s *base;
};
```

After being assigned to the `function` field of a `timer_list`, we need to figure out where this might flow and where `saa7146_buffer_timeout` might eventually be called. Some searching through the code reveals that the `function` field never has its address taken, and is only ever assigned to the local variable `fn` in the core kernel function `__run_timers`.

```
// kernel/timer.c
static inline void __run_timers(tvec_base_t *base)
{
   struct timer_list *timer;

   ...
   while (...) {
      ...

      while (...) {
         void (*fn)(unsigned long);
         unsigned long data;
```

```
            timer = list_entry(head->next,struct timer_list,entry);
            fn = timer->function;
            data = timer->data;

            set_running_timer(base, timer);
            detach_timer(timer, 1);

            ...
            fn(data);
            ...
        }
    }

    set_running_timer(base, NULL);
}
```

__run_timers then repeatedly pulls timer_list's off of lists, and calls the timer->function function pointer with the timer->data value. This exposes the design intent of the timer_list structure: whatever is stored in the function field of a timer_list will be subsequently called with the data field of that same timer_list.

Then, when saa7146_buffer_timeout is called here then it is only be called with whatever the data field is in the timer_list the pointer to saa7146_buffer_timeout is stored in. In the cases of vbi_init and video_init, the data field points to the vbi_q or video_q fields, respectively, of a saa7146_vv structure. Both of these queue fields are of type saa7146_dmaqueue, which, recall, is the same type that saa7146_buffer_timeout expects. If saa7146_buffer_timeout is only called with these queues for its data parameter, then the cast it performs is safe.

So, is it the case that saa7146_buffer_timeout might be called with a parameter other than these two queues, vbi_q and video_q? There are two ways this could arise. First, the data field could be

assigned another value (of a possibly different type) between the call to `vbi_init`/`video_init` and `__run_timers`. Second, the `function` field could be called somewhere else outside `__run_timers` with a parameter other than the `data` field of that `timer_list`.

As it turns out, in general for a `timer_list` **both** of these cases are possible. Fortunately, neither case happens to impact any `timer_list` which contains `saa7146_buffer_timeout` in the `function` field.

Typically, the `function` and `data` fields of a `timer_list` are written at the same time, shortly after the `timer_list` is created. Because of the behavior of `__run_timers`, it does not do much good to set the `function` field without setting the `data` field as well, and vice versa. These writes do not impact the data passed to `saa7146_buffer_timeout`, as the `function` field will be changed to a value other than `saa7146_buffer_timeout`. However, there are two cases in Linux where the `data` field is written without any corresponding write to the `function` field. An example is `tlclk_interrupt`.

```
// drivers/char/tlclk.c
static irqreturn_t tlclk_interrupt(...)
{
   ...
   int_events = inb(TLCLK_REG6);
   ...

   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);
   } else {
      ...
   }
   ...
```

```
   return IRQ_HANDLED;
}
```

Under certain circumstances `tlclk_interrupt` will change the `data` field of the global `switchover_timer` variable to a non-pointer integer value. However, `switchover_timer` cannot alias the `timer_list`'s which contain `saa7146_buffer_timeout` — `switchover_timer` is a statically allocated `timer_list`, while the timers manipulated by `vbi_init` and `video_init` are embedded in an outer `saa7146_dmaqueue` structure.

The second case where `saa7146_buffer_timeout` might receive a value other than the two input queues occurs when the `function` field of a `timer_list` is invoked with a different argument 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 (though at runtime this is where the overwhelming majority of timers are invoked). The other four follow similar patterns, with `ctnetlink_del_conntrack` as one example.

```
// 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 instead, exploiting knowledge that these two pointers are in fact identical; the `ct->timeout.data` field is a parent pointer back to `ct` itself. As with `switchover_timer` from the previous example, in this case we know `ct->timeout` cannot alias the timers from `vbi_init` and `video_init`, as the two timers are embedded in different types of structures. Thus, the call to `ct->timeout.function` cannot invoke `saa7146_buffer_timeout`.

1.3 Analyzing Casts

The great majority of casts in Linux require reasoning on par with that for the `saa7146_buffer_timeout` cast, and for many the reasoning is far more complex.

The central concept behind our analysis is to emulate this reasoning process, to model as closely as possible model the arguments that a human reader will make when determining the correctness of the code. A human reader has the advantage of a good understanding (hopefully!) of how the code works, allowing them to more easily wade through the code to find the relevant portions. An automatic analysis has the advantage of far more computational power, allowing it to go through all the code that could possibly be relevant, and to ideally analyze it with sufficient precision to arrive at the same conclusion as the human.

An important idea falling out of this central concept is that, while proving the correctness of a cast may require tens or hundreds of reasoning steps, each of those steps is fairly small and self contained. Most

systems code needs to be well-written to be easy to understand, to maintain and to update. As such, most of the code's invariants and other abstractions will be simple, and will follow from each other with simple reasoning steps. Problems arise from either the sheer scale of the system, or from the use of strange or overly complex abstractions.

We divide the casting analysis according to the main types of reasoning required to determine the correctness of a cast, each of which is reflected in the previous example.

- Function pointers: to know the possible values of `data` when `saa7146_buffer_timeout` is called, we first need to know *where* `saa7146_buffer_timeout` might be called. We can get this information by finding the places where the function is mentioned, and then follow it through any assignments until we get to the indirect calls in `__run_timers` and `ctnetlink_del_conntrack`.

  Alternatively, we can start at the indirect calls themselves, and follow assignments to the function pointer backward until we find the possible functions it could be referring to. The complete set of possible callers are those indirect call sites whose target can be followed backwards to `saa7146_buffer_timeout`.

- Escaping locations: extending the function pointer analysis concept, we can track any location as it is assigned from one location to another, going forwards or backwards to find out where it might have originated or where it might be consumed.

  Within the `ctnetlink_del_conntrack` function, we determined that the indirect call `ct->timeout.function(...)` could not target `saa7146_buffer_timeout` because the two `ct->timeout` fields cannot refer to the same timer as `vv->vbi_q` in `vbi_init`.

If the call was instead through some `ptimer->function(...)`, where `ptimer` has type `timer_list*`, we could still determine whether `ptimer` can point to `&vv->vbi_q` by following back along the assignments to `ptimer` to see where it came from. Alternatively, we could take assignments of `&vv->vbi_q` and follow them forward to see where the address of that field might go to.

- Polymorphic data correlations: of all the possible values of the `data` and `function` fields of a `timer_list`, only a few values of `data` can be associated with a particular `function`, and vice versa. By looking at all the assignments to these fields, we can determine that when the `function` is `saa7146_buffer_timeout`, the only possible values for `data` are those which were set by `vbi_init` and `video_init`. When `__run_timers` invokes the `function` field and calls `saa7146_buffer_timeout`, the only possible data arguments are these two values.

- Cast propagation: finally, we can check the safety of the cast in `saa7146_buffer_timeout` by propagating backwards from the cast to make sure all the possible values of `data` are of type `saa7146_dmaqueue`. With the function pointer and escape analyses we determine the only caller of `saa7146_buffer_timeout` is `__run_timers`, and with the polymorphic data analysis we determine the possible values of `data` when `saa7146_buffer_timeout` is invoked at that call site. These two values `vv->vbi_q` and `vv->video_q` are of type `saa7146_dmaqueue`, thus the cast is safe.

Within each of these main analyses, there is further subdivision possible, again along the lines of how we expect a human reader to reason about the code. At a broad level, each of the analyses is composed from three pieces:

- A small set of *abstraction types*. Abstractions are the basic facts which the reader builds up as they try to prove the correctness of a cast or whatever property they care about. Abstraction types are the patterns which abstractions tend to follow, e.g. "Field f of structure *str* is never `NULL`." There are few abstraction types relevant to each analysis because, for code readability and maintainability, programmers tend to reuse existing abstractions wherever possible.

- For each type of abstraction, a set of independent, heuristic sub-analyses to infer that abstraction from the code and from other inferred abstractions. These encode the individual reasoning steps the reader makes, how new facts can be learned from existing facts and from reading the code, e.g. "Field f of structure *str* is never `NULL` if it is only assigned the values of other non-`NULL` fields."

  We call these sub-analyses heuristic because they target and are built around specific patterns used over and over in the code. These sub-analyses are generalized to handle as many cases as possible, but no single one can handle all the ways by which the abstraction could be inferred. Using multiple independent sub-analyses that infer the same sort of information but target wholly different coding patterns greatly increases the total coverage.

- For each type of abstraction, a small set of annotations to handle cases too strange or non-standard to be covered by the heuristics, and too important to simply be ignored. Most problems we are interested in are complex enough to make it extremely difficult to build a fully automatic, scalable analysis that handles all cases with sufficient precision. On the other hand, an easy to

build analysis can handle many cases but not enough, leaving an overwhelming annotation burden for the remainder.

We try to split the difference with analyses that are just complex enough to handle the great majority of cases, leaving annotations for the small remainder.

In the remainder of this section we give a quick overview of each of the main analyses used in this thesis. Complete descriptions of the analyses, their abstractions and sub-analyses are given in their respective chapters. The annotations we used are covered within Chapter 9.

1.3.1 Function Pointer Analysis

The function pointer analysis constructs a coarse but useful upper bound on the possible targets of each indirect call.

- Two types of abstractions are in use. First, which program variables and structure fields might have their values later used as the target of a particular indirect call? Second, for these indirect calls, what are the functions which might be assigned to those variables and fields?

- Sub-analyses follow assignments between variables and fields. If there is some 'x = y;' and the value of 'x' might be used by a later call, the value of 'y' might also be used by that call. If 'y' is a specific function, that function may be a specific target of the call.

- Annotations are needed to model cases involving more than just variables and fields, such as storing a function in a heap-allocated array.

1.3.2 Escape Analysis

The escape analysis takes a program location and finds all the other places that location either might escape to or might have escaped from via a series of assignments.

- One main abstraction is used: for two locations l_0 and l_1, is it possible that l_1 might have the same value as l_0, due to a series of assignments from l_0 to l_1?
- Sub-analyses follow assignments to construct a forward or backward closure from the initial location. Locations vary in coarseness ("Field f of variable x" is a location, and so is "field f of all values of type *str*"), and different sub-analyses vary the coarseness of the search they perform.
- Annotations are only needed to cut off the search when it propagates to locations irrelevant to analysis. In particular, we abstract away the data structures underlying the primitive memory allocators (e.g. `kmalloc`).

1.3.3 Polymorphic Data Analysis

The polymorphic data analysis finds the possible correlations between particular function pointers and particular bits of program data when they are stored in polymorphic structures like `timer_list`. These structures are ubiquitous in C systems code.

- Two abstractions are used. First, which calls can use a polymorphic relationship between a function pointer and data stored in the same structure? Second, for those polymorphic relationships, what are the particular possible correlations that might exist?

- We need just one sub-analysis to find the polymorphic relationships for each call by comparing the function pointer target of the call with the arguments passed in. For `__run_timers`, the two are fields of the same `timer_list` structure. Getting a precise bound on the correlations is far more difficult, and requires several sub-analyses targeted to the initialization patterns used by the bulk of these structures.

- Annotations are needed for any polymorphic structure whose initialization does not match the fairly narrow patterns covered by the sub-analyses.

1.3.4 Casting Propagation Analysis

The casting analysis constructs a proof for the correctness of each cast by enumerating the possible reasons that cast could be correct (due to a type invariant, type constraint in the caller, etc.) and recursively trying, in parallel, to prove that any one of those reasons must hold.

- The abstraction types describe the proofs and subproofs that can be made about the types of values in the program. "x always has type *str* at entry to foo" is a subproof, as are "the first argument to function call i in foo always has type *str*" and "field f of type $xstr$ always has type *str*".

- As with the polymorphic data analysis, there are many possible ways each subproof might be proved, often with the help of one or more other subproofs. Each of these ways corresponds to a different sub-analysis.

- Few annotations are needed. Casts we can't prove correct with

the information and techniques we have are rejected as possibly unsafe.

1.4 Soundness and Completeness

The analyses described here are incomplete — they approximate the possible behaviors of the code in places they do not understand it. This approximation is nearly sound, though; besides the exceptions described below, it does completely characterize the possible behaviors, and if it marks a downcast as safe, that downcast is safe.

The only source of unsoundness in the analyses as designed (the abstractions and sub-analyses described in Section 1.3) are that they assume the program is type safe along its execution path to the point of the cast being analyzed. If the program has a type safety violation and that first violation is a bad downcast, the analysis will find it. However, if there is a first violation which is **not** a bad downcast, but is some other type safety violation, the analysis may or may not find any later bad downcasts.

More important are a few sources of unsoundness that creep in during the implementation of the analysis. These are the following:

- The annotations described in Section 1.3 are trusted. Since they describe pieces of the code the sub-analyses do not understand, we cannot check if they are actually correct. If the annotations have bugs and do not characterize the code correctly, we will not detect these bugs.

- The abstractions inferred by the sub-analyses are designed to be correct by construction (i.e. the sub-analyses themselves should be sound and always infer sound information), and we do not have

an independent way to verify this. If the sub-analyses contain bugs, we will not detect these bugs.

- Most of the sub-analyses are summary-based, only analyzing and generating information for a single function at a time. Analysis can time out on a small fraction of functions, and these timeouts can affect the soundness of the analysis results for those functions (some sub-analyses, in particular those used for the final casting analysis itself, produce sound information even in the presence of timeouts). These timeouts affect less than .5% of the functions in the Linux version we analyzed.

With these caveats, out of 28767 downcasts in Linux, our analysis is able to prove the safety of 21637, or 75.2% (this population does not include every downcast in Linux; for details see Section 9.6). We have gone through these results extensively both to work on proving previously missed casts, and to look for bogus proofs, those which are incorrect due to one of the above factors. Out of several hundred examined we have found a few bogus proofs (all caused by timeouts), so while these bogus proofs exist, they are an extremely small fraction of the number of proved casts.

1.5 Contributions

The main contributions of this thesis are summarized as follows:

- We developed the Saturn infrastructure, which is based around a new logic programming language for summary-based program analysis (Chaper 2). All our analyses are written using this language.

- We specify an exact memory model for the behavior of a function or loop body (Chapter 3). This model soundly and accurately considers interprocedural aliasing and side effects of calls (Section 5.3), and is used by most of our analyses.

- We specify a function pointer analysis for identifying indirect call targets, which is self-contained and achieves fairly accurate results (Chapter 4).

- We specify an escape analysis for following values forward or backward through the program, which is largely demand driven and offers a highly tunable level of precision (Chapter 5).

- We describe the use of data correlations to model the behavior of polymorphic structures, and specify an analysis to discover these correlations (Chapter 6).

- We specify an analysis to prove the safety of downcasts to structure types, in the presence of polymorphism and other complex heap behavior (Chapter 7).

- We describe in detail the results of these analyses, including where the analyses fall short when applied to Linux, how these failures are addressed using annotations, and code patterns in Linux falling entirely outside the purview of the analyses (Chapter 9).

Chapter 2

Saturn Infrastructure

All the analyses we will describe were built using the Saturn program analysis infrastructure, for which we given an overview in this chapter. Saturn is a highly scalable infrastructure supporting the precise analysis of C code, which can run efficiently on a computer cluster and is designed for the sorts of analyses we will use in this thesis: analyses which examine a small piece of the program at a time (usually a single function), and use heuristic techniques to infer abstractions from that piece and from the other abstractions that have already been inferred. (Saturn as described here was developed based on the original version of Saturn by Xie and Aiken [31]).

These analyses have two important characteristics.

- Abstractions are simple and compact. It should be easy to concisely specify an abstraction.
- There may be many independent techniques for inferring the same abstraction. It should be easy to write and to tweak these techniques for the target code base.

To accomodate these characteristics, Saturn analyses use a very

simple execution model. All abstractions and other data are expressed as predicates over strings, integers, bools, and other datatypes. All analysis inference techniques and other logic are expressed as rules in a logic programming language over these predicates.

With this model, specifying an abstraction just requires one to define one or a few predicates and the types of their arguments. Specifying a new inference technique requires one to add one or more rules and maybe some intermediate predicates. The result is an extremely compact analysis; the entirety of the analyses in this project, from the initial syntax trees to the final proofs of correct casts, are written in 6031 non-comment non-blank lines of code.

The remainder of this chapter is organized as follows. In Section 2.1 we describe the logic programming language. In Sections 2.2 and Section 2.3 we describe how the language interfaces with the front end syntax trees and any back end constraint solvers, respectively. Finally, in Section 2.4 we describe how the logic programs are used to perform interprocedural analysis.

2.1 Logic Programming Language

The logic programming language used by Saturn analyses is Calypso, which was designed specifically for Saturn (out in the real world, Calypso is a small, irregularly shaped moon orbiting the planet Saturn). Calypso is a fairly standard Datalog-like language [28], with extensions for interfacing with external databases and constraint solvers, and to allow an easy mixture of eager and demand-driven analysis.

In this thesis we cover only the portion of Calypso syntax used in later chapters to specify the various Saturn analyses. We use a couple of running examples through the following subsections to illustrate various

concepts.

2.1.1 Values and Types

Facts in Calypso are predicates over primitive values. Calypso values and predicates are strongly and statically typed, such that every value and variable has a type known during compilation which must match the signature of the predicate in which it is used.

Types in Calypso are the primitive `string`, `int`, `bool`, analysis-defined datatypes, and opaque types defined by solvers and other packages external to the analyses (see Section 2.3). Datatypes are declared with the `type` keyword.

The following example defines a datatype for the possible nodes in some graph with a distinguished source and sink. Possible values of type `node` include `source`, `sink`, `point{"foo"}` and `point{"bar"}`.

```
type node ::= source | sink | point{string}.
```

Datatypes can be polymorphic over other types. In the following example, the variable `T` may be any type; the type of a list of integers is `list[int]`, and contains such values as `nil` and `cons{1,cons{2,nil}}`. Note also that type `list[T]` is recursive, defined in terms of itself.

```
type list[T] ::= nil | cons{T,list[T]}.
```

Type aliases can be created with the `=` operator. In the following example, `list[int]` and `intlist` may be used interchangeably.

```
type intlist = list[int].
```

Types can be declared but not defined by omitting the `::=` and `=` operators. In this case the type's contents are left abstract, and the type may be completely opaque if it is defined within a package.

The grammar for type identifiers, variables, types and type declarations is as follows:

id	:	string beginning with lowercase letter
var	:	string beginning with uppercase letter
anyvar	:	'_' (an unnamed variable)
type	::=	*var* | *id* | *id* '[' var_0 ',' ... ']'
declval	::=	*id* | *id* '{' $type_0$ ',' ... '}'
typedecl	::=	'type' *type* '::=' $declval_0$ '|' ... '.'
	|	'type' *type* '=' *type′* '.'
	|	'type' *type* '.'

2.1.2 Predicates

Predicates in Calypso are declared with the `predicate` keyword, specifying the number and type of arguments the predicate takes. The following predicate defines the space of graph edges between values of type `node`.

```
predicate graph_edge(X:node, Y:node).
```

All known facts about the program are encoded using predicates. For each predicate declaration, at any given time there is a finite set of known *instances* of that predicate — that predicate name with particular values for its arguments. For the `graph_edge` predicate a few example instances are `graph_edge(source,point{"foo"})` and `graph_edge(point{"bar"},sink)`. The job of the Calypso rules is to infer new predicate instances from the known instances.

As with types, Calypso predicates can be polymorphic. The following predicate can relate a list of items to the individual items in the list, for any type of list.

```
predicate list_mem(L:list[T], M:T).
```

The grammar for predicate declarations is as follows (though we will extend this slightly in Section 2.1.4):

$$\begin{array}{rcl} \textit{predarg} & ::= & \textit{var} \text{ ':' } \textit{type} \\ \textit{preddecl} & ::= & \text{'predicate' } \textit{id} \text{ '(' } \textit{predarg}_0 \text{ ',' ... ')' '.'} \end{array}$$

2.1.3 Rules

Rules in Calypso follow the same basic syntax as Prolog and Datalog. These are Horn clauses of the form X:-Y, Z. indicating that if predicate instances Y and Z both hold, predicate instance X also holds. The arguments to X, Y and Z may refer to variables, which will be *bound*, or unified with, all the corresponding arguments in the known predicate instances.

Consider the following example, which computes all nodes reachable over `graph_edge` predicate instances from the `source` node.

```
predicate source_reach(X:node).
source_reach(X) :- graph_edge(source,X).
source_reach(Y) :- source_reach(X), graph_edge(X,Y).
```

The first rule indicates that any node connected by a graph edge from `source` is reachable from `source`. For each edge from `source`, variable `X` is bound to the target of that edge, and then the new instance of `source_reach` is created with that bound value of `X`.

The second rules indicates that any node connected by a graph edge from any node reachable from `source` is itself reachable from `source`. For each node in `source_reach`, bind `X` to that node, and then for any node reachable from that `X`, bind `Y` to the target and add `source_reach` for that `Y`.

Besides the simple rule queries used in the bodies of the rules deriving `source_reach`, there are three other types of clauses that can be used: unification, negation, and collection clauses.

Unification clauses are specified with the `=` operator between two values, unifying them and binding variables on one side to the corresponding values on the other. If the two sides are not unifiable, such as `source=point{_}`, the unification fails and the rule's target predicate instance is not added. If we replace the rules for `source_reach` with the following rules, we will compute only the `point` nodes reachable from `source` via other `point` nodes.

```
source_reach(X) :-
   graph_edge(source,X), X=point{_}.
source_reach(Y) :-
   source_reach(X), graph_edge(X,Y), Y=point{_}.
```

Unification clauses can be negated by replacing `=` with `\=`, yielding a clause that succeeds if the two sides are **not** unifiable. Replacing `=` with `\=` in the above rules will compute only the non-`point` nodes reachable from `source` through non-`point` nodes.

Negation queries, specified with the $\sim$ operator, do not bind any variables, but are tests that hold only if there are no predicate instances which match the arguments to the negation. The following rule computes all the nodes which have an incoming edge but are not reachable from the source node.

```
predicate not_reach(X:node).
not_reach(X) :- graph_edge(_,X), ~source_reach(X).
```

Collection queries, specified with the `\/` operator, are used to combine all predicate instances matching its arguments using a special collection operator. The following rule computes a list containing all nodes

reachable from the source node. The `list_all` operator constructs a list `L` from all values `X` which are bound by some `source_reach` predicate instance.

```
predicate all_reach(L:list[node]).
all_reach(L) :- \/source_reach(X):list_all(X,L).
```

The grammar for Calypso rules is as follows:

$$
\begin{array}{rcl}
val & ::= & var \mid int \mid bool \mid string \\
 & \mid & id \mid id \text{ `\{'} val_0 \text{ `,'} \ldots \text{ `\}'} \\
predval & ::= & id \text{ `('} val_0 \text{ `,'} \ldots \text{ `)'} \\
ruleval & ::= & predval \mid val_0 \text{ `=' } val_1 \mid val_0 \text{ `\textbackslash=' } val_1 \\
 & \mid & \text{`}\sim\text{' } predval \mid \text{`}\backslash/\text{' } predval \text{ `:' } predval \\
rule & ::= & predval \text{ `:-' } ruleval_0 \text{ `,'} \ldots \text{ `.'}
\end{array}
$$

2.1.4 Modes and Execution Model

Logic programming languages normally use either an *eager* or *demand-driven* execution model (also known as bottom-up and top-down). Datalog variants tend to be eager, computing all predicate instances that can possibly be derived from the rules. Prolog variants tend to be demand-driven, computing only the predicate instances necessary to prove a particular goal, or target predicate instance.

The execution model used for Calypso programs is mostly eager. For the `source_reach` predicate in the previous section, given a set of `graph_edge` instances representing the graph, the rules will be applied exhaustively to find all nodes in the graph reachable from `source`.

The demand-driven alternative would be to compute `source_reach` for particular nodes referenced in other rules. For example, if there

was a rule `X :- ..., source_reach(sink).` we would try to compute `source_reach` specifically for `sink` with a backtracking search backwards along `graph_edge` instances from `sink`.

Trying this search using the two rules which derive `source_reach` is likely to be incredibly wasteful, revisiting the same nodes over and over as it finds new paths to them backwards from `sink`. It is possible to write an efficient demand-driven graph search using lists of the visited and frontier nodes in the search, but the result is cumbersome and negates much of the reason for using a logic programming language in the first place.

There are, however, cases where we want to be demand-driven. Consider the `list_mem` predicate from earlier, for which we can define rules as follows:

```
list_mem(L,X) :- L=cons{X,_}.
list_mem(L,X) :- L=cons{_,TAIL}, list_mem(TAIL,X).
```

Applying these rules eagerly results in constructing `list_mem` for all possible lists, a process which does not terminate.

We can specify that `list_mem` must be computed in a demand-driven fashion using a *mode* for the `list_mem` predicate. A mode extends the predicate declaration to specify which arguments to the predicate must already be known before that predicate can be computed. Each predicate has a set of input arguments specified with the `in` keyword and a set of output arguments specified with the `out` keyword (`out` is the default if neither keyword is used). The execution model is then: for particular values of the input arguments, what are the possible values of the output arguments?

In the case where there are no input arguments, this becomes 'what are the possible values of all the arguments,' the eager execution model from before.

For `list_mem`, we want to be sure that the `list_mem` instances are only computed for particular lists mentioned during the evaluation of other rules. The declaration of `list_mem` with the desired mode is as follows:

```
predicate list_mem(in L:list[T], out M:T).
```

Now, when there is a rule which includes as a clause `list_mem(XL,X)` where, during evaluation, `XL` is bound to a particular list such as `cons{1,cons{2,nil}}`, then the rules for `list_mem` will be applied but only after binding `L` to that particular list. If there is a rule such as `list_mem(_,X)` then a compile-time error will be generated because `_` could be any value.

The last concern with modes are *determinism constraints*. For particular values of the input arguments to a predicate, how many different values of the output arguments can there be? In general, there could be zero (the predicate failed), one or more outputs for any given input. A determinism constraint, specified with the `succeeds` keyword, specifies a range for the possible number of outputs, and after execution terminates the constraints are checked and warnings generated for predicates which violated them.

The possible constraints are `zero`, `once`, or `many` (one or more), and multiple constraints can be specified in which case the union of their ranges is used.

Some examples of determinism constraints are below. `list_mem` may succeed any number of times, while fetching the last member of a list succeeds at most once, and fetching the length of a list always succeeds once.

```
predicate list_mem(in L:list[T], out M:T)
   succeeds [zero,many].
```

```
predicate list_last(in L:list[T], out M:T)
   succeeds [zero,once].
predicate list_length(in L:list[T], out LEN:int)
   succeeds [once].
```

The updated grammar for predicate declarations is as follows:

$$
\begin{array}{rcl}
\mathit{succ} & ::= & \text{`zero'} \mid \text{`once'} \mid \text{`many'} \\
\mathit{succlist} & ::= & \text{`succeeds'} \text{`['} \mathit{succ}_0 \text{`,'} \ldots \text{`]'} \\
\mathit{predarg} & ::= & [\text{`in'}|\text{`out'}]? \ \mathit{var} \text{`:'} \ \mathit{type} \\
\mathit{preddecl} & ::= & \text{`predicate'} \ \mathit{id} \text{`('} \mathit{predarg}_0 \text{`,'} \ldots \text{`)'} \ \mathit{succlist}? \ \text{`.'}
\end{array}
$$

2.1.5 Sessions

During software analysis, Calypso programs are run not over the entire source of the target code base, but rather over the code for individual functions, global variables, or types. With the Calypso features shown so far, there is no way for the Calypso analyses of different functions to communicate with one another — the Calypso program will do some processing on the function's syntax, maybe print some output, and then terminate. To allow interprocedural analysis, the program run on one function needs to be able to receive information about the runs on other functions, and in turn to pass new information about the function it is analyzing on to the runs on other functions.

The model we use to allow this communication is a *session*, a map that stores predicate instances and persists across the entire interprocedural analysis. The keys in a session are *session instances*, an identifier with zero or more value arguments (the same syntactic structure as predicate instances). The values in a session are sets of predicate instances.

Calypso programs are free to query the predicate instances in a

session, and to add new instances to the session. After the program terminates, all predicate instances computed which were not stored in a session are thrown away; these are intermediate results which will not be used by other runs in the interprocedural analysis.

To illustrate how sessions work, we revisit the graph search example in a program analysis context. Consider the problem of identifying functions that always abort the program, such as `abort`, `exit`, and, in Linux, `panic`. It is useful to know these functions because control never returns to points after their call sites; if we account for this fact during analysis we can more accurately reason about code where aborts occur.

Additionally, these functions often have *wrappers* in different parts of a code base, user-defined functions which always terminate the program by calling a primitive abort function or another wrapper. Instead of finding these by hand we can infer them, looking for functions where there is no path from the entry point to exit point which does **not** pass through a call to a function known to abort. We will write an analysis to compute the aborting functions and store that information in the `abort_function` session.

```
session abort_function(FN:string) containing [abort].
predicate abort().
```

This declaration specifies a session map named `abort_function` from function names to a set of `abort` predicate instances (`containing` specifies the possible predicates that can appear in a session). Because `abort` takes no arguments, there is only a single possible instance of it, so for any function `FN`, `abort_function(FN)` will contain either no instances, or the single instance `abort()`. In the former case, `FN` is not known to always abort (it is possible that it might, however), and in the latter case calls to `FN` will always abort.

To compute `abort_function`, we need a model of the current function's control flow graph. Points in the control flow graph are represented with the abstract type `pp` (for "program point"). This is a datatype filled in by whichever analysis constructs the control flow graph).

```
type pp.
```

The edges and the entry/exit/call points of the control flow graph are modelled with the predicates below, along with the name of the current function.

```
predicate cfg_edge(P0:pp,P1:pp).
predicate cfg_entry(P:pp).
predicate cfg_exit(P:pp).
predicate cfg_call(P:pp,FN:string).
predicate current_function(FN:string).
```

We introduce a demand-driven auxiliary predicate `abort_call`, which indicates for any point in the CFG whether that is a call to an abort function — it either specifies `abort` explicitly, or it calls some function for which the `abort_function` session contains the `abort` predicate. The operator `->` is used to access the contents of a session.

```
predicate abort_call(in P:pp).
abort_call(P) :-
   cfg_call(P,"abort").
abort_call(P) :-
   cfg_call(P,FN), abort_function(FN)->abort().
```

Finally, we compute `abort_function(FN)->abort()` with a graph reachability analysis. Predicate instance `entry_reach(P)` indicates that point `P` is reachable from the function entry point over a path that does not pass through an abort call.

```
predicate entry_reach(P:pp).
entry_reach(P1) :-
   cfg_entry(P0), ~abort_call(P0), cfg_edge(P0,P1).
entry_reach(P1) :-
   entry_reach(P0), ~abort_call(P0), cfg_edge(P0,P1).

abort_function(FN)->abort() :-
   current_function(FN), cfg_exit(P), ~entry_reach(P).
```

For information on the execution model used for interprocedural analysis, see Section 2.4. The grammar for session declarations is as follows:

$$
\begin{array}{rcl}
\mathit{containlist} & ::= & \text{`contains'} \text{ `[' } \mathit{id}_0 \text{ `,' ... `]'} \\
\mathit{sessarg} & ::= & \mathit{var} \text{ `:' } \mathit{type} \\
\mathit{sessdecl} & ::= & \text{`session'} \mathit{id} \text{ `(' } \mathit{sessarg}_0 \text{ `,' ... `)'} \mathit{containlist}? \text{ `.'}
\end{array}
$$

We extend the *predval* nonterminal used in the grammar for rules as follows to handle queries and updates on session contents.

$$
\begin{array}{rcl}
\mathit{predval} & ::= & \mathit{id} \text{ `(' } \mathit{val}_0 \text{ `,' ... `)'} \\
& | & \mathit{id} \text{ `(' } \mathit{val}_0 \text{ `,' ... `)' `->' } \mathit{id} \text{ `(' } \mathit{val}_0 \text{ `,' ... `)'}
\end{array}
$$

2.2 Frontend AST

Saturn uses the Cil infrastructure [20] as a frontend to generate the abstract syntax tree, or AST, for each function, type and global variable in the target program. These trees are converted to predicates, and the resulting predicates are stored in several sessions for use by later analyses. The sessions storing these AST predicates are as follows:

```
cil_body(FN:string).
cil_comp(C:string).
```

```
cil_glob(G:string).
cil_init(G:string).
```

The session `cil_body(FN)` contains the AST for the definition of function `FN` — the names and types of its arguments and locals, and all its control flow, assignments and calls. The session `cil_comp(C)` contains the AST for the composite struct or union type `C`, while `cil_glob(G)` contains the AST for the declaration of a global variable `G` and `cil_init(G)` contains the AST for its static initializer, if it has one.

The syntax predicates themselves are over opaque types for the different kinds of AST nodes, such as `c_instr`, `c_lval`, and `c_exp` for assign/call instructions, lvalues and expressions, respectively. The predicates relate one node in the AST to its children; for example, a memory access appears as an instance `cil_lval_mem(LV,ME,OFF)`, relating the dereference lvalue `LV` with the expression which was dereferenced `ME` and the field/array offset `OFF` from `*ME`. This structure directly reflects the Cil syntax tree.

2.2.1 AST Session Analysis

Different Calypso programs may be run on each of the functions, globals, or types in the program. Each program specifies which kind of AST it should run over by specifying the name of that session with the `analyze` keyword.

```
analyze session_name("cil_body").
```

This directive indicates to the interprocedural analysis that to run this program, some `cil_body(FN)` session instance needs to be picked, and all the predicate instances stored in it loaded as the initial known instances.

2.2.2 AST Control Flow Graphs

Almost all Saturn analyses that analyze functions work from a low level control flow graph, rather than the structured if/while/etc. statements of the AST. When analyzing the `cil_body` session, a control flow graph is generated from the function's syntax, encoding the entirety of the function's execution with the following low-level types and predicates:

- `type pp.` An abstract type for program points in the function's execution.
- `predicate cfg_entry(P:pp).` The unique entry point of the currently analyzed function.
- `predicate cfg_exit(P:pp).` The unique exit point of the currently analyzed function.
- `predicate cfg_branch(P:pp,P0:pp,P1:pp,E:c_exp).` If the point of execution is at `P`, control will transfer to either `P0` or `P1`, depending on whether the value of expression `E` is nonzero or zero, respectively.
- `predicate cfg_call(P0:pp,P1:pp,I:c_instr).` If the point of execution is at `P0`, the function call indicated by `I` is executed and control transfers to `P1`.
- `predicate cfg_set(P0:pp,P1:pp,LV:c_lval,E:c_exp).` If the point of execution is at `P0`, the location indicated by lvalue `LV` is assigned the value of expression `E`.

The main property maintained for these control flow graphs is they do not contain loops. The local memory analysis which will be used by many analyses (Section 3.3) constructs an exact model of a function's

behavior, and as such does not terminate on control flow graphs containing loops. Loops in the AST's flow are removed by converting them to tail recursive functions. Client analyses will treat these new tail recursive loop functions as regular source functions, and analyze them independently from the source function they were originally contained in. An example of this *loop splitting* is as follows:

```
void foo(int *buf)
{
  for (i = 0; i < 100; i++) {
    buf[i] = 0;
  }
}

foo
{
  i = 0;
  call(foo_loop);
}

foo_loop
{
  if (i < 100) {
    buf[i] = 0;
    i++;
    call(foo_loop);
  }
}
```

The Linux kernel version we analyze has 91,384 source functions and 34,409 loops. The analyses described in future chapters do not distinguish between the CFGs generated for these two categories, and we refer to these collectively as 'functions' throughout.

2.3 Constraint Solver Interface

Many of our analyses need to be path-sensitive — that is, to consider the conditions under which paths are taken and particular program points are reached. Performing a path-sensitive analysis requires us to do constraint solving to determine which paths through a function are actually feasible or not. Logic programming languages are ill-suited to constraint solving, and as such we have designed Calypso to allow programs to easily interface with external solvers and other *packages*.

A package in Calypso is just a set of opaque types and predicates. When the Calypso program queries one of the predicates, instead of looking for rules which can derive that predicate the Calypso interpreter will call into the underlying solver to get the query's result.

For example, a package for operations on finite sets includes the following definitions (and some others). The `set[T]` type is opaque, and is represented internally as a sorted list (the sets Calypso programs generate typically don't get big enough to make a faster implementation worthwhile). The various predicates either get the members of the set, or create new sets from existing sets and values; a collection operator `set_all` can be used in a collection like `list_all` to construct a set from all predicate instances matching some query.

```
type set[T].
predicate set_member(in S:set[T], out X:T)
   succeeds [zero,many].
predicate set_empty(out S:set[T])
   succeeds [once].
predicate set_insert(in S:set[T], in X:T, out NS:set[T])
   succeeds [once].
collection set_all(in X:T, out NS:set[T]).
```

The most important package in Saturn defines predicates for constructing boolean formulas and passing them on to a SAT solver. These formulas are used to determine path feasibility in the target program. As with the set logic package, the boolean formula package includes an opaque type `bval[T]` for formulas over unconstrained boolean variables of type `T`, and various predicates to construct them: `bool_g` constructs a true or false formula, `id_g` constructs an unconstrained leaf formula for some variable, and these are combined with the `and`, `or` and `not` predicates or the `and_all` and `or_all` collections. Finally, the satisfiability of any formula `G` is queried with `sat(G)`. Internally, using the `sat` predicate converts the formula to CNF and calls into an underlying SAT solver.

```
type bval[T].
predicate bool_g(in B:bool, out G:bval[T])
   succeeds [once].
predicate id_g(in X:T, out G:bval[T])
   succeeds [once].
predicate and(in G0:bval[T], in G1:bval[T], out G:bval[T])
   succeeds [once].
predicate or(in G0:bval[T], in G1:bval[T], out G:bval[T])
   succeeds [once].
predicate not(in NG:bval[T], out G:bval[T])
   succeeds [once].
predicate sat(in G:bval[T]).
collection and_all(in G:bval[T], out MG:bval[T]).
collection or_all(in G:bval[T], out MG:bval[T]).
```

2.4 Interprocedural Analysis

Interprocedural analysis in Saturn is a fixpoint computation. When a Calypso program runs on a source function, it both queries and updates

the predicate instances in various session instances. If the Calypso run on one function queries one session instance, and a later run on another function updates that same session instance, the first function will have to be reanalyzed. For example, consider running the Calypso program from Section 2.1.5 to generate the `abort_function` session on the following C program:

```
void foo() { bar(); }
void bar() { abort(); }
```

If we run the program on `bar` first, we will add the predicate instance `abort_function("bar")->abort()`. When we then run on `foo` we will also add `abort_function("foo")->abort()` and reach the desired fixpoint.

However, if we run the program on `foo` first, then since the session `abort_function("bar")` is initially empty we will **not** add the instance `abort_function("foo")->abort()`. When we then run on `bar` we will add `abort_function("bar")->abort()`, changing a session instance the analysis of `foo` depended on and necessitating a reanalysis of `foo`. Reanalyzing `foo` will add `abort_function("foo")->abort()`, again reaching the desired fixpoint.

We express interprocedural analysis in Saturn as a worklist algorithm using the following definitions.

- $s \in Sess$: A session, a map from session instances to sets of predicate instances.
- $k \in SessInst$: A session instance, a key in some session. The map isess : $SessInst \Rightarrow Sess$ gets the session each instance is a key in (this is just the identifier portion of the instance — an instance cannot be a key for multiple sessions).

- $p \in CLP$: A compiled Calypso program. The map psess : $CLP \Rightarrow Sess$ indicates the session over which the program should be run, as specified by an `analyze session_name(...)` directive in the program (Section 2.2).

The worklist algorithm is as follows:

1. Take as input the initial sessions $S \in 2^{Sess}$ for the program syntax and any other previously computed information, and a set of programs $P \in 2^{CLP}$ to cofixpoint. For each program $p \in P$, the session psess(p) must be in S.

2. Let $W \in 2^{CLP \times SessInst}$ be the worklist. The items in W are Calypso programs and the session instances to run them over (typically `cil_body(FN)` for some function `FN`). For each $p \in P$ and $k \in$ keys(psess(p)) add (p, k) to W.

3. Let $D \in SessInst \Rightarrow 2^{CLP \times SessInst}$ be the dependency map indicating, for each session instance, which worklist items queried that instance while they were running. D initially maps all instances to the empty set.

4. Remove some (p, k) from W. Run p over k by loading all predicate instances in (isess(k))[k] and fully executing all rules in p. For any session instance k_r which was queried, add (p, k) to $D[k_r]$. Then, for any session instance k_w which had new predicates added, for all $(p', k') \in D[k_w]$ add (p', k') to W.

5. Repeat step 4 until W is empty.

2.4.1 Implementation and Distributed Analysis

Calypso programs take a long time to run on multi-million line programs, due mostly to the complexity of the analyses but also to the need to handle each function separately and the slow speed of the Calypso interpreter vs. a language compiled to native code. This is more than offset by distributing the interprocedural analysis over a computer cluster, a process which is made easier by the highly parallel nature of the interprocedural worklist algorithm.

All the analysis runs we will report were performed on a 100 core cluster, though we normally use only half the cluster for any single analysis. One core is a server that manages the worklist and controls all access to the session data (each session is stored on disk as a Berkeley DB database [1]). The remainder are workers that takes jobs from the server (a Calypso program and session contents `s[k]`) and runs the Calypso interpreter, querying the server for the contents of other session instances and informing the server of any updates to make to the sessions.

Since all session data I/O goes through the server core, the server can easily keep track of which items should be in the worklist W and dependency map D. This also runs the risk of making the distributed analysis I/O bound, if the server gets bogged down answering data requests from workers. In practice, using 50-70 cores on a switched gigabit network we typically get between 30% and 60% efficiency (how much of the maximum possible time each worker spent running the interpreter and not doing server I/O), depending on the Calypso program being run. For more detailed results, see Section 8.2.

Chapter 3

Memory model

All the analyses we will describe are based on a common memory model for C, which we present in this chapter. This includes:

- A global, flow-insensitive view of memory for reasoning about the flow of data at a broad level through the entire program.
- A local, path-sensitive view of memory for reasoning precisely about the flow of data through the body of a single procedure or loop body.

Different analyses will use one of the two views of memory, depending on their precision. To allow these analyses to communicate with one another, the two views of memory use a common representation of heap locations, access paths (Section 3.1). Sections 3.2 and 3.3 describe the two views of memory and how they model the assignments and control flow of the program.

3.1 Access Paths

The `t_trace` type represents an access path which refers, at any particular point in the program, to a single heap location. Traces are defined using the additional types `t_root` for program variables, and `scalar` for arithmetic formulas over integers and traces.

```
type t_trace ::=
    root{R:t_root}
  | drf{T:t_trace}
  | fld{T:t_trace,F:string,C:string}
  | rfld{T:t_trace,F:string,C:string}
  | index{T:t_trace,Y:t_type,I:scalar}
  | empty
  .

type t_root ::=
    arg{A:int}
  | glob{G:string}
  | local{L:string}
  | temp{TMP:string,WHERE:string}
  | return
  .

type scalar ::=
    s_const{N:int}
  | s_trace{T:t_trace}
  | s_unop{OP:unop,V:scalar}
  | s_binop{OP:binop,V0:scalar,V1:scalar}
  .
```

The interpretation of traces is as follows:

- `root{R}`: The stack- or statically-allocated location of a program variable, indicated by `R`.

- `drf{T}`: The value of trace `T`. The location `*x` for a local variable `x` is `drf{root{local{"x"}}}`.

- `fld{T,F,C}`: Field `F` by structure type `C` of trace `T`. The location `x->f` for a local variable `x` of type `str*` is:

  `fld{drf{root{local{"x"}}},"f","str"}`.

- `rfld{T,F,C}`: The 'reverse' field `F` by structure type `C` of trace `T`. A common pattern in systems code is to pass around the addresses of internal structure fields, and recover the base pointer to the structure through pointer arithmetic, e.g. the `container_of` macro in Linux:

  ```
  // include/linux/kernel.h
  #define container_of(ptr, type, member) ({                \
      const typeof( ((type *)0)->member ) *__mptr = (ptr);  \
      (type *)( (char *)__mptr - offsetof(type,member) );})
  ```

  The location `container_of(x, str, f)` for a local variable `x` is `rfld{drf{root{local{"x"}}},"f","str"}`. `rfld` and `fld` are inverse functions; `rfld{fld{T,F,C},F,C}` is equivalent to `T`.

- `index{T,Y,I}`: The element at index `I` of array trace `T`. `Y` indicates the type of the array elements (`char`, `int`, some structure type, etc.) and hence the stride used. The location `x[5]` for a stack-allocated array `int x[10]` is:

  `index{root{local{x}},y_int,s_const{5}}`

  The location `x[5]` for a local variable `int *x` is:

  `index{drf{root{local{x}}},y_int,s_const{5}}`

- `empty`: An empty trace. Traces containing `empty` rather than a `root{R}` are *relative* and describe an offset between traces, rather than a particular heap location.

Every non-relative trace `T` is derived via a chain of accesses from a `t_root` program variable `R`. We say here that `T` is *rooted* at `R`, use the `trace_root` predicate to find the root of any trace `T` if it exists.

```
predicate trace_root(in T:t_trace, out R:t_root)
   succeeds [zero,once].
```

Traces can be taken apart and put back together with the `trace_sub` and `trace_compose` predicates.

```
predicate trace_sub(in T:t_trace,
                    out ST:t_trace, out RT:t_trace)
   succeeds [many].
predicate trace_compose(in ST:t_trace, in RT:t_trace,
                        out T:t_trace)
   succeeds [once].
```

`trace_sub` computes all the *subtraces* `ST` of `T`, with `RT` a relative trace indicating the offset from `TS` to `T`. For example, applying `trace_sub` to `fld{drf{root{local{"x"}}},"str","f"}` (the C expression `x->f`) yields:

```
trace_sub(...,fld{drf{root{local{"x"}}},"str","f"},empty)
trace_sub(...,drf{root{local{"x"}}},fld{empty,"str","f"})
trace_sub(...,root{local{"x"}},drf{fld{empty,"str","f"}})
```

`trace_compose` computes the inverse of `trace_sub`, producing the original trace `T` from the subtrace `ST` and offset `RT`.

3.2 Global memory model

The global memory model is a mechanism for quickly finding the reads and writes in the program involving particular sets of access paths. For example, where are the places in the program where a particular field is read from? What if that field is embedded in a particular outer struct or accessed through a particular global variable? We describe these sets of access paths with *trace locations*.

3.2.1 Trace Locations

We use the `t_trace` representation of an access path to describe sets of locations within the program. The type `t_trace_loc` pairs a trace with some function, global variable or structure type, and describes all heap locations matched by that access path and function/global/structure at any point in the program.

```
type t_trace_loc ::=
    tl_func{FN:string,T:t_trace,CXT:t_call_context}
  | tl_glob{T:t_trace}
  | tl_comp{C:string,RT:t_trace}.

type t_call_context ::=
    cxt_any
  | cxt_call{PFN:string,PI:c_instr,PCXT:t_call_context}.
```

The interpretation of trace locations is as follows:

- `tl_func{FN,T,CXT}`: Access path `T` rooted at an argument, return variable, local or temporary of function `FN`, where `FN` is called within a context described by `CXT`. `cxt_any` describes all calling contexts for `FN`, while `cxt_call{PFN,PI,PCXT}` describes only those contexts when `FN` is called through instruction `PI` of

PFN (with calling context PCXT for PFN). (c_instr is the type of syntactic call instructions)

- tl_glob{T}: Access path T rooted at some global variable, as that global is used within any function in the program.
- tl_comp{C,RT}: Access path RT relative to all values used as structure type C in any function in the program. RT does not have a root and is based at the value empty.

Analogous to traces, trace locations have predicates trace_loc_sub and trace_loc_compose to compute sublocations and offset traces and compose these back together. These simply apply the trace_sub or trace_compose predicates to the single trace included in the location.

```
predicate trace_loc_sub(in L:t_trace_loc,
                        out SL:t_trace_loc, out RT:t_trace)
   succeeds [many].
predicate trace_loc_compose(in SL:t_trace_loc, in RT:t_trace,
                            out L:t_trace_loc)
   succeeds [once].
```

At any point in the execution of the program, each heap location that is reachable from stack or static variables (i.e. isn't leaked) can be represented using one or more trace locations. Consider the following toy example:

```
typedef struct str {
  int f;
} str;

void main()
{
  foo();
```

```
}

void foo()
{
  str x;
}
```

When the program enters the body of `foo`, field `x.f` within `foo` is represented by each of the following trace locations:

- `tl_func{"foo",fld{local{"x"},"f","str"},`
  `cxt_call{"main","call0",cxt_any}}`

  Identifies the field `f` of `x` when `foo` is called by `main` (the label used for the call within `main` is `call0`).

- `tl_func{"foo",fld{local{"x"},"f","str"},cxt_any}`

  Identifies the field `f` of `x` when `foo` is called by any function.

- `tl_comp{"str",fld{empty,"f","str"}}`

  Identifies the field `f` of any value of type `str`.

An analysis could choose to abstract `x.f` with any of these locations. The concern of the memory model is to capture all the assignments and function calls that could read or write from each of these locations. Given an arbitrary trace location, where are the points in the program where matching heap locations could be written with new data or read into another location?

3.2.2 Treatment of assignments and calls

For the purposes of the global memory model, assignments and call sites both introduce data flow, with some function context information attached. These flow edges are of the form `L0 -> L1` with `L0` and

`L1` both trace locations. The following rules describe how flow edges are generated for assignments, call site argument bindings and call site return value assigns:

- An assignment `x = y` in a function `F` where `XT` and `YT` are the access path traces for `x` and `y`, respectively, introduces flow:

  ```
  tl_func{F,YT,cxt_any} -> tl_func{F,XT,cxt_any}
  ```

- A call `G(y,...)` with identifier `C` in a function `F` where `AT` is the trace for the argument within `G` (i.e. `drf{root{arg{_}}}`) and `YT` is the trace for `y` introduces flow:

  ```
  tl_func{F,YT,cxt_any} ->
     tl_func{G,AT,cxt_call{F,C,cxt_any}}
  ```

- A call `x = G(...)` with identifier `C` in a function `F` where `XT` is the trace for `x` introduces flow:

  ```
  tl_func{G,drf{root{return}},cxt_call{F,C,cxt_any}} ->
     tl_func{F,XT,cxt_any}
  ```

- A global static initializer `x = y` (`y` must be an address expression, and both `x` and `y` either global variables or are at field/array offsets from global variables). where `XT` and `YT` are the traces for `x` and `y`, respectively, introduces flow:

  ```
  tl_glob{YT} -> tl_glob{XT}
  ```

If the type of the assignment, argument, or return value is a structure (e.g. `x = y;` where `x` and `y` are structures, not pointers to structures) then flow is instead introduced for each field of that structure, transitively following any sub-structures.

Additionally, for any assignment, argument, or return value whose source is the address of a structure's inner field, we need to add additional flow edges that 'back out' the field accesses using `rfld`. This implicit flow accounts for the C programming practice of passing around internal fields of a structure and later backing them out with the `container_of` macro. For a flow edge with source `fld{S,F,C}` and target `T`, add a new flow edge with source `S` and target `rfld{T,F,C}`. For example, the assignment `x = &y->g.h;` leads to the following edges:

```
fld{fld{drf{y},g,gstr},h,hstr} -> drf{x}
fld{drf{y},g,gstr} -> rfld{drf{x},h,hstr}
drf{y} -> rfld{rfld{drf{x},h,hstr},g,gstr}
```

3.2.3 Matching Trace Locations

Given a particular trace location `L`, we need to find all the flow edges where the source or target of the edge *matches* `L`; either `L`'s value is copied to another location, or `L` receives a new value from another location. A match between `L` and the source or target `XL` occurs when the sets of access paths described by `L` and `XL` intersect. The location match relation is defined as follows:

- Locations `tl_func{FN,T,CXT0}` and `tl_func{FN,T,CXT1}` match if either `CXT0` or `CXT1` is a prefix of the other.
- For a trace `T` rooted at a global variable, `tl_glob{T}` matches location `tl_func{FN,T,CXT}` for any `FN` and `CXT`.

- `tl_comp{C,RT}` matches any location `L` with a sublocation `SL` such that `trace_loc_compose(SL,RT,L)` holds.

3.3 Local Memory Model

The local view of memory provides a precise path-sensitive model of a loop-free function body's possible behaviors — the branches it might take, assignments it might make, and functions it might call. It does this by computing the possible program states at each point in the function's execution in terms of the program state at entry to the function, in a demand-driven fashion.

When modelling the behavior of the branches and assignments, the local memory analysis is exact; it loses no information and is maximally precise. The only source of imprecision comes in reasoning about the behaviors of other functions, which can influence the local analysis in two ways: which access paths at entry to the function might alias, and which locations might be written to by calls in the function. These two cases are abstracted away into interface predicates to be defined externally from the local memory analysis (Sections 3.3.5 and 3.3.6).

Program states are represented using traces for memory locations, scalars for pointer and integer values, and a new type `g_guard` for boolean conditions. Type `g_guard` is that of boolean formulas over comparisons between scalars and unconstrained bits (see Section 3.3.5), defined using the `bval` abstract type from Section 2.3.

```
type scalar_cmp ::=
    sc_cmp{OP:binop,V0:scalar,V1:scalar}
  | sc_eqz{V:scalar}
  | sc_bit{B:scalar_bit}.

type g_guard = bval[scalar_cmp].
```

Wherever they appear, traces always refer to the state at function entry. For example, when referred to by another trace, scalar, or guard at some program point `P` (of type `pp`), the trace `drf{root{arg{0}}}` refers to to the initial value of the function's first argument, regardless of any changes to that argument at points prior to `P`.

There are two core predicates in the local memory analysis which we will focus on, `guard` and `val`.

```
predicate guard(in P:pp, out G:g_guard)
   succeeds [once].

predicate val(in P:pp, in T:t_trace,
              out V:scalar, out G:g_guard)
   succeeds [many].
```

For any program point `P` in the function, `guard` indicates the condition `G` under which that point will be executed. For any program point `P` and trace `T`, `val` indicates the possible values `V` of the location represented by `T` at point `P` under any execution, and the conditions `G` under which the location holds those values.

Conceptually, the possible executions of the function are satisfying assignments for the terms in the `g_guard` and `scalar` values. For each such satisfying assignment the traversed path through the function is the set of points whose `guard` condition holds under the assignment, and at each point in that path the value of each trace is given the `val` condition holding for that point and trace in the assignment.

Two invariants relate the `guard` and `val` predicates. These invariants ensure that along any execution path through the function each trace has exactly one value at each point in the path. First, the conditions for the different values of a trace are pairwise disjoint after conjunction with the point guard: if for some `P` and `T`, where `guard(P,G)`,

`val(P,T,V0,G0)`, `val(P,T,V1,G1)` and `V0 \= V1`, then `G` $\wedge$ `G0` $\wedge$ `G1` is unsatisfiable. Second, the disjunction of the conditions for the different values of a trace implies the point guard: if for some `P` and `T`, where `guard(P,G)` and `\/val(P,T,_,XG):or_all(XG,MG)`, then `G` $\wedge$ `MG` is equivalent to `MG`.

To illustrate how the `guard` and `val` predicates work, consider the following example:

```
void foo(int *x, int *y, int q)
{
   if (q != 0) {
      x = y;
   }
   else {
   }
   *x = 0;
}
```

We comment this example with the guards as follows. `*q` represents the trace `drf{root{arg{2}}}`, i.e. the initial value of `q` regardless of any writes to `q` (and similarly for other traces in future examples). Inside the `if` statement the guard indicates the conditions along which the true and false branches are executed, reverting to the condition `true` afterwards.

```
void foo(int *x, int *y, int q)
{
   // true
   if (q != 0) {
      // *q != 0
      x = y;
      // *q != 0
   }
   else {
```

```
      // *q == 0
   }
   // true
   *x = 0;
   // true
}
```

The values and associated conditions for `x`, `y`, `*x` and `*y` are as follows. Conditions which are simply `true` are omitted.

```
void foo(int *x, int *y, int q)
{
   // x -> *x, y -> *y, *x -> **x, *y -> **y
   if (q != 0) {
      // x -> *x, y -> *y, *x -> **x, *y -> **y
      x = y;
      // x -> *y, y -> *y, *x -> **x, *y -> **y
   }
   else {
      // x -> *x, y -> *y, *x -> **x, *y -> **y
   }
   // x -> *x [*q == 0], x -> *y [*q != 0], y -> *y
   // *x -> **x, *y -> **y
   *x = 0;
   // x -> *x [*q == 0], x -> *y [*q != 0], y -> *y
   // *x -> 0 [*q == 0], *x -> **x [*q != 0]
   // *y -> 0 [*q != 0], *y -> **y [*q == 0]
}
```

When the assignment `*x = 0` is considered, either `*x` or `*y` is updated to hold the value zero, depending on whether the branch `q != 0` was taken and `x` was updated to refer to `*y`. Note that this example assumes that `x` and `y` are not aliased, that updating `*x` cannot change the value of `*y` and vice versa. We will first define the rules computing `guard` and `val` without considering aliasing and the effects of function

calls (Sections 3.3.2, 3.3.3 and 3.3.4), and will extend this definition to handle these two cases in Sections 3.3.5 and 3.3.6.

3.3.1 Secondary Predicates

Before getting to the definitions of `guard` and `val`, we need to define some secondary predicates which are both used while computing `guard` and `val` and are also helpful for client analyses of the local memory analysis.

First, we can use the following predicates to get the value of a syntax lvalue or expression at points in the CFG.

```
lval(in P:pp, in LV:c_lval, out T:t_trace, out G:g_guard)
   succeeds [many].
eval(in P:pp, in E:c_exp, out V:scalar, out G:g_guard)
   succeeds [many].
beval(in P:pp, in E:c_exp, out BG:g_guard, out G:g_guard)
   succeeds [many].
```

For any program point `P` and syntax lvalue `LV`, `lval` gives the possible traces `T` the lvalue can refer to and associated condition `G` (if the lvalue contains no dereferences or array accesses, there will be a single trace and the condition will be true). For any program point `P` and syntax expression `E`, `eval` gives the possible values `V` the expression can have and associated conditions `G`. `beval` functions in the same way as `eval`, but computes the condition `BG` under which the value `V` is non-zero.

`lval`, `eval`, and `beval` are implemented as descents down the syntax tree for the lvalue/expression, querying the `val` predicate for all accesses to memory. For example, consider evaluating the `(q != 0)` expression in the sample code above. This expression will be encoded with the following predicates.

- `cil_exp_binop(ECMP,b_ne,EQ,E0,_)`: `ECMP` is a `!=` binary operation expression over expressions `EQ` and `E0`.
- `cil_exp_lval(EQ,LVQ)`: Expression `EQ` reads the value of the lvalue `LVQ`.
- `cil_lval_var(LVQ,QVAR,QOFF)`, `cil_off_none(QOFF)`: Lvalue `LVQ` represents `QVAR`, the variable for the third argument, without any field/array offset.
- `cil_exp_const(E0,C0)`, `cil_const_int(C0,_,0)`: `E0` is an expression for the integer constant zero.

The relevant rules and their descriptions for deriving the instances matching `beval(P,ECMP,_,_)` from these input predicates are as follows.

1. Get the condition for a comparison binary operation by evaluating the left and right sides with `eval` and using the `id_g` predicate (Section 2.3) to produce a boolean formula for that leaf condition.

```
beval(P,E,BG,G) :-
   cil_exp_binop(E,OP,LE,RE,_),
   eval(P,LE,LV,LG), eval(P,RE,RV,RG),
   and(LG,RG,G), id_g(sc_cmp{OP,LV,RV},BG).
```

2. Evaluate an lvalue expression by getting the trace the lvalue refers to with `lval`, and getting the current value of that trace with `val`.

```
eval(P,E,V,G) :-
   cil_exp_lval(E,LV),
   lval(P,LV,T,LG), val(P,T,V,VG), and(LG,VG,G).
```

3. Get the trace a variable lvalue refers to using the `var_trace` auxiliary predicate, which associates a `t_root` value with each program variable.

```
lval(P,LV,root{R},G) :-
   cil_lval_var(LV,X,OFF), cil_off_none(OFF),
   var_trace(X,R), bool_g(true,G).
```

4. Get a constant `scalar` value for each integer constant expression.

```
eval(P,E,s_const{N},G) :-
   cil_exp_const(E,C), cil_const_int(C,_,N),
   bool_g(true,G).
```

Applying these rules from the bottom up to the predicate instances for `q != 0` gives us the following predicates, in turn:

```
eval(P,E0,s_const{0},true)
lval(P,LVQ,root{arg{2}},true)
eval(P,EQ,s_trace{drf{root{arg{2}}}},true)
beval(P,ECMP,sc_cmp{b_ne,s_trace{drf{root{arg{2}}}},
                    s_const{0}},true)
```

We get a single `beval` result for the outer `q != 0` expression, for the condition where the trace `drf{root{arg{2}}}` (the initial value of `q`) is not zero. If `q` could have multiple values at the point of the `q != 0`, (e.g. there was an assignment to `q` along some but not all incoming paths), there would be multiple `beval` results.

Two additional secondary predicates are not used directly by the memory analysis itself, but by client analyses.

```
inst_trace(in I:c_instr, in P:pp, in CT:t_trace,
           out V:scalar, out G:g_guard)
```

```
   succeeds [many].
convert_trace(in P:pp, in T:t_trace,
              out V:scalar, out G:g_guard)
   succeeds [many].
```

All traces, scalars, and guards computed by the local memory analysis are local to the currently analyzed function; they are expressed in terms of that function's input state. To do interprocedural analysis a client must be able to translate traces, scalars, and guards expressed for one function into corresponding values expressed for another function. For a call instruction `I` at point `P`, predicate `inst_trace` translates a trace `CT` relative to the entry state of the callee into its possible values `V` relative to the entry state of the caller. This is a simple translation: `drf{arg{A}}` traces are translated to the value of argument `A` at the call site, and for other `drf{T}`, `T` is translated to new possible traces `T'`, and the `val` predicate used to get the values of those `T'` traces at point `P`.

Predicate `convert_trace` is the same as `inst_trace`, except for the special casing of `drf{arg{A}}` traces. This has the effect of converting a trace relative to some point `P` (the representation used by the global memory analysis) into a value relative to the entry state of the current function.

3.3.2 Computing `guard(P,G)`

The `guard` for a point is the condition on the function's input state where that point is reached during the function's execution. Reachability is over the control flow graph which we defined in Section 2.2.2. We define a new predicate `cfg_edge` as the union of all the branch/call/set edges within the CFG.

```
predicate cfg_edge(P0:pp, P1:pp).
cfg_edge(P,P0) :- cfg_branch(P,P0,_,_).
cfg_edge(P,P1) :- cfg_branch(P,_,P1,_).
cfg_edge(P0,P1) :- cfg_call(P0,P1,_).
cfg_edge(P0,P1) :- cfg_set(P0,P1,_,_).
```

Now, for each CFG edge there is an associated condition under which the first point jumps to the second point, which we will store in the `edge_cond` predicate. For branches, we merge the possible values of the expression `E` into a single condition `MG`, and use `MG` as the condition on the true branch and `!MG` on the false branch. For non-branches, the `edge_cond` condition is just `true`.

```
predicate edge_cond(P0:pp, P1:pp, G:g_guard).

predicate one_beval(in P:pp, in E:c_exp, out G:g_guard)
   succeeds [many].
one_beval(P,E,MG) :- beval(P,E,BG,G), and(BG,G,MG).

predicate merge_beval(in P:pp, in E:c_exp, out G:g_guard).
   succeeds[once].
merge_beval(P,E,MG) :- \/one_beval(P,E,G):or_all(G,MG).

edge_cond(P0,P1,G) :-
   cfg_branch(P0,P1,_,E), merge_beval(P,E,G).
edge_cond(P0,P1,G) :-
   cfg_branch(P0,_,P1,E), merge_beval(P,E,NG), not(NG,G).
edge_cond(P0,P1,G) :-
   cfg_edge(P0,P1), ~cfg_branch(P0,_,_,_), bool_g(true,G).
```

For any path in the CFG to an intermediate point, that path will be taken during execution of the function iff the conjunction of the `edge_cond` edges along that path holds at entry. The guard at a point `P` is the disjunction of this conjunction over all paths to `P`.

An equivalent definition is that the guard at P is the disjunction, for all predecessor points P0, of the conjunction of edge_cond(P0,P,EG) and guard(P0,G0) (which is encapsulating the condition for all paths reaching P0). This value is easily computed.

```
predicate gmerge(in P:pp, out G:g_guard).
guard(P,MG) :- \/gmerge(P,G):or_all(G,MG).

gmerge(P,G) :-
   cfg_entry(P), bool_g(true,G).
gmerge(P1,G) :-
   edge_cond(P0,P1,EG), guard(P0,G0), and(G0,EG,G).
```

3.3.3 Computing Edge Guards

One tricky thing about the val predicate is that the conditions in val and in guard, while computed each using the other, are otherwise independent from one another. If for some P and T, guard(P,G) and val(P,T,_,VG), VG does not have to imply G. This was shown in the earlier example with val, where x -> *y under condition true at the end of the if statement's true branch, and this condition was refined to *q != 0 only at the join point after the if.

While it would be valid to have the above condition VG always imply G, enforcing this may cause a blowup in the size of the conditions in val and resulting performance degradation. So we would like to keep the conditions in val as general and simple as possible while still ensuring the conditions VG of each trace are pairwise disjoint after intersection with G.

We use a refinement predicate eguard(P0,P1,G,NG) to simplify the val guards. eguard computes a guard NG from G such that, where guard(P0,G0), guard(P1,G1), and edge_cond(P0,P1,EG), NG $\wedge$ G1 $\Leftrightarrow$

`G` $\wedge$ `G0` $\wedge$ `EG`. For the example discussed above, the refinement at the `if` from `true` to `*q != 0` satisfies this property.

For points `P1` which are not join points and have only the single incoming edge, it suffices to choose `NG` = `G` since `G1` itself is `G0` $\wedge$ `EG`. For join points, we choose `NG` = `G` $\wedge$($\neg$ `G1` $\vee$(`G0` $\wedge$ `EG`)), which satisfies the biconditional and in practice allows considerable simplification in the `val` guards.

```
predicate eguard(in P0:pp, in P1:pp, in G:g_guard,
                 out EG:g_guard)
   succeeds [once].

predicate cfg_join(P:pp).
cfg_join(P) :- cfg_edge(P0,P), cfg_edge(P1,P), P0\=P1.

eguard(P0,P1,G,G) :- ~cfg_join(P1).
eguard(P0,P1,G,EG) :- cfg_join(P1),
   guard(P0,G0), guard(P1,G1), edge_cond(P0,P1,ECG),
   not(G1,GNOT), and(G0,ECG,GAND), or(GNOT,GAND,GOR),
   and(G,GOR,EG).
```

3.3.4 Computing `val(P,T,V,G)`

As with `guard`, we will compute `val` by accumulating all the possible values along each incoming edge to a point, and taking their disjunction. This is performed with the `vmerge` predicate, analogous to `gmerge`. Since, again, traces in the local memory model are expressed in terms of the function's entry state, at the entry point each trace `T` always points to trace `drf{T}`.

```
predicate vmerge(in P:pp, in T:t_trace,
                 out V:scalar, out G:g_guard).
val(P,T,V,NMG) :-
   vmerge(P,T,V,_), \/vmerge(P,T,V,G):or_all(G,MG).
```

```
vmerge(P,T,s_trace{drf{T}},G) :-
   cfg_entry(P), bool_g(true,G).
```

In this section, recall, we will compute `val` while only considering the effects of direct assignments, and not aliasing or call site side effects. Thus, we only need to care about the `cfg_set` edges and the lvalues they update. If the lvalue contains dereferences, it might refer to one of several traces depending on the previous assignments in the function. Recall the earlier example, where the update to `*x` could update either the trace `*x` or `*y`, depending on whether `x` had been assigned `y` earlier. We capture the possible direct updates to traces with the `assign` predicate. As with `val`, a trace might be updated to different values at a point under disjoint conditions.

```
predicate assign(P:pp, T:t_trace, V:scalar, G:g_guard).
assign(P,T,V,G) :- cfg_set(P,_,LV,E),
   lval(P,LV,T,G0), eval(P,E,T,G1), and(G0,G1,G).
```

Generating the `assign` predicate on the example program yields the following assignments:

```
void foo(int *x, int *y, int q)
{
   if (q != 0) {
      x = y; // x := *y
   }
   else {
   }
   *x = 0;   // *x := 0 [*q == 0], *y := 0 [*q != 0]
}
```

Now, all we have to do to compute `vmerge` is, for each edge and trace, to account for the cases where the trace either is assigned a new

value, or is not assigned at all and keeps its old value. The following two rules deal with these cases. These are the only rules which will change when we extend `val` to handle aliasing and call site side effects.

```
vmerge(P1,T,V,NG) :- cfg_edge(P0,P1),
   assign(P,T,V,G), eguard(P0,P1,G,NG).

vmerge(P1,T,V,NG) :- cfg_edge(P0,P1),
   \/assign(P0,T,_,AG):or_all(AG,MAG), not(MAG,NMAG),
   base_val(P0,T,V,VG), and(NMAG,VG,G), eguard(P0,P1,G,NG).
```

3.3.5 Handling Aliasing

The handling of `val` we have shown in the example thus far assumes that `x` and `y` are not aliased, that `*x` and `*y` refer to different heap locations. To be sound, `val` needs to do better, to consider possible aliasing between traces when handling assignments. We introduce an unconstrained boolean variable `alias(*x,*y)` that is true iff `x` aliases `y` (this is equivalent to the guard `*x == *y`).

When this aliasing might occur, the computed `val` predicates are as follows. The only differences vs. the use of `val` without aliasing are the possible values of `*x` and `*y` at the CFG exit point.

```
void foo(int *x, int *y, int q)
{
   // x -> *x, y -> *y, *x -> **x, *y -> **y
   if (q != 0) {
      // x -> *x, y -> *y, *x -> **x, *y -> **y
      x = y;
      // x -> *y, y -> *y, *x -> **x, *y -> **y
   }
   else {
      // x -> *x, y -> *y, *x -> **x, *y -> **y
   }
```

```
   // x -> *x [*q == 0], x -> *y [*q != 0], y -> *y
   // *x -> **x, *y -> **y
   *x = 0;
   // x -> *x [*q == 0], x -> *y [*q != 0], y -> *y
   // *x -> 0 [*q == 0 || alias(*x,*y)]
   // *x -> **x [*q != 0 && !alias(*x,*y)]
   // *y -> 0 [*q != 0 || alias(*x,*y)],
   // *y -> **y [*q == 0 && !alias(*x,*y)]
}
```

To determine whether two traces are aliased, the local memory analysis uses the predicate `trace_alias`, which takes as input two traces and produces the condition under which they alias (or fails if they can never alias).

```
predicate trace_alias(in T0:t_trace, in T1:t_trace,
                      out G:g_guard)
   succeeds [zero,once].
```

Predicate `trace_alias` is used as an oracle by the memory analysis; it is not actually defined by the memory analysis, and can be instantiated differently by different clients, depending on their level of need for soundness and precision. The `trace_alias` rules we will use for subsequent analyses are described in Section 5.3.1.

Despite leaving the aliasing rules abstract for now, there are some general principles reasonable rules will follow, pertaining to the previous and future examples:

- Any trace `T` always aliases itself.
- Stack and global variables which never have their address taken never alias any other trace.

To account for aliasing within assignments, instead of using the direct update predicate `assign`, we use a demand-driven predicate `assign_alias` which indicates the condition where a trace `T` aliases some trace `XT` that is directly written (`T` might be equal to `XT`).

```
predicate assign_alias(in P:pp, in T:t_trace,
                       out V:scalar, out G:g_guard).
assign_alias(P,T,V,G) :-
   assign(P,XT,V,XG), trace_alias(T,XT,AG), and(AG,XG,G).
```

Predicate `assign_alias` has the same signature as `assign` but handles even indirect assignments through aliasing, so to capture possible aliasing while computing `vmerge` we just replace `assign` with `assign_alias`:

```
vmerge(P1,T,V,EG) :- cfg_edge(P0,P1),
   assign_alias(P0,T,V,G), eguard(P0,P1,G,EG).

vmerge(P1,T,V,MG) :- cfg_edge(P0,P1),
   \/assign_alias(P0,T,_,AG):or_all(AG,MAG), not(MAG,NMAG),
   val(P0,T,V,VG), and(NMAG,VG,G), eguard(P0,P1,G,EG).
```

3.3.6 Handling Calls

The state of memory within the function can be affected by either assignments or calls. As shown previously, we will model the regular assignments exactly, using guards to case split depending on whether the left side of the assignment aliases various traces. For some calls we will do the same thing, adding `assign` instances to exactly model their side effects. In general, though, we cannot model the effects of calls with such precision. In a large program such as Linux most calls can have side effects on thousands of heap locations. It is not feasible

to even fully represent this set of locations, let alone describe exactly what actions the call takes on them.

Thus, we introduce a second interface predicate `trace_clobber` which, along the same lines as `trace_alias`, allows client analyses to specify the points at which a call might update a trace to some unknown new value. We use a new type of trace `uc_sum{I,P,T}` for the unconstrained result of the call `I` at point `P` clobbering the value of trace `T`. While this new trace is still determined by the function's entry state (provided the program is deterministic), we have no idea which location it refers to without delving into the behavior of the call at `I`.

```
predicate trace_clobber(in I:c_instr, in P:pp, in T:t_trace).

type t_trace ::= ... | uc_sum{I:c_instr,P:pp,T:t_trace}.
```

As with `trace_alias`, we leave the rules for `trace_clobber` abstract for now, along with the cases where we will add `assign` instances for calls, and revisit them in Section 5.3.2.

With clobbering, we now need to consider three cases instead of two for the value of a trace after a CFG edge. The trace may be updated via `assign_alias`, may keep its old value, or may be clobbered by a call. The latter two cases are encapsulated by the `base_val` predicate; `T` is clobbered iff the edge leaving `P` is a call for which `trace_clobber` holds on `T`.

```
predicate base_val(in P:pp, in T:t_trace,
                   out V:scalar, out G:g_guard).

base_val(P,T,V,true) :- icall(P,_,I),
   trace_clobber(I,P,T), V = s_trace{uc_sum{I,P,T}}.
base_val(P,T,V,G) :- icall(P,_,I),
   ~trace_clobber(I,P,T), val(P,T,V,G).
base_val(P,T,V,G) :- ~icall(P,_,_), val(P,T,V,G).
```

When we compute `vmerge`, the case where `T` is updated by a direct assignment takes precedence over clobbering; adding an `assign` edge for a call simply provides more detailed information than the `trace_clobber`. Thus, we use `assign_alias` in the same way, and just substitute `base_val` for `val` when considering the condition where `T` is not aliased with any directly updated trace.

```
vmerge(P1,T,V,EG) :- cfg_edge(P0,P1),
   assign_alias(P0,T,V,G), eguard(P0,P1,G,EG).

vmerge(P1,T,V,MG) :- cfg_edge(P0,P1),
   \/assign_alias(P0,T,_,AG):or_all(AG,MAG), not(MAG,NMAG),
   base_val(P0,T,V,VG), and(NMAG,VG,G), eguard(P0,P1,G,EG).
```

Chapter 4

Function Pointer Analysis

Pretty much any sound interprocedural analysis requires a complete call graph, which gives an overapproximation of the possible targets of each indirect call. For each call to some arbitrary expression, what are the actual functions to which that function could refer at runtime? Function pointers tend to be used in programs in a straightforward manner, assigning them to plain variables, arrays, or fields, but rarely using them in a more complex fashion, such as casting a function pointer to some other type or creating a heap-allocated buffer of function pointers. We want an analysis that can characterize the usual behaviors precisely, and fall back to manual annotation for the complex behaviors.

The overall effectiveness of this analysis is determined by the amount of complex behavior it cannot capture that is **actually present** in the analyzed program, and hence the annotation burden it requires. As we will see in Section 9.2, this burden is small — 46 annotations for 100,000 indirect call edges — and the analysis presents a good trade/off between its own complexity and the annotations it requires.

4.1 Example

Recall the example from Section 1.2.

```
// drivers/media/common/saa7146_fops.c
void saa7146_buffer_timeout(unsigned long data)
{
   ...
}

// drivers/media/common/saa7146_video.c
static void video_init(struct saa7146_dev *dev,
                       struct saa7146_vv *vv)
{
   vv->video_q.timeout.function = saa7146_buffer_timeout;
   ...
}

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

We are ultimately interested both in what are the possible callers of `saa7146_buffer_timeout`, and what are the possible callees of the

indirect call `fn(data)` in `__run_timers`. By identifying the possible targets for the call `fn(data)`, as well as the possible targets for all other indirect calls in the kernel, we will have complete call graph information which lets us answer both questions conservatively.

We can find the targets for `fn(data)` just by following `fn` back through the assignments in the program. The value of `fn` is the same as whatever it was written with earlier, `timer->function`. So what are the possible values of `timer->function`? We could answer this by following `timer` itself back through the (very complex) code in the kernel which maintains lists of pending timers, back to the corresponding calls to `add_timer` which scheduled the timer, but there is an easier way. Ignore `timer` itself, and just look for assignments to the `function` field of **any** `timer_list` in the program. This is easy to do and will find the assignment of `saa7146_buffer_timeout` to `vv->video_q.timeout.function` within `video_init`.

For all other writes to the `function` field, if that write is of a particular function then that function is now a potential target of the `fn(data)` call in `__run_timers`, and if that write is **not** of a particular function, we will need to continue following the written value back through the assignments in the program. An example of where this occurs is `inet_csk_init_xmit_timers`.

```
// net/ipv4/inet_connection_sock.c
void inet_csk_init_xmit_timers(struct sock *sk,
                       void (*retransmit_handler)(unsigned long),
                       void (*delack_handler)(unsigned long),
                       void (*keepalive_handler)(unsigned long))
{
   struct inet_connection_sock *icsk = inet_csk(sk);

   init_timer(&icsk->icsk_retransmit_timer);
   init_timer(&icsk->icsk_delack_timer);
```

```
   init_timer(&sk->sk_timer);

   icsk->icsk_retransmit_timer.function = retransmit_handler;
   icsk->icsk_delack_timer.function     = delack_handler;
   sk->sk_timer.function                = keepalive_handler;

   icsk->icsk_retransmit_timer.data =
      icsk->icsk_delack_timer.data =
         sk->sk_timer.data  = (unsigned long)sk;

   icsk->icsk_pending = icsk->icsk_ack.pending = 0;
}
```

The `function` of a `timer_list` may be any of the function pointers passed to `inet_csk_init_xmit_timers`; we have to look at all call sites to this function. `inet_csk_init_xmit_timers` is called in two places, by `dccp_init_xmit_timers` and `tcp_init_xmit_timers`.

```
static void dccp_write_timer(unsigned long data);
static void dccp_keepalive_timer(unsigned long data);
static void dccp_delack_timer(unsigned long data);

// net/dccp/timer.c
void dccp_init_xmit_timers(struct sock *sk)
{
   inet_csk_init_xmit_timers(sk, &dccp_write_timer,
                             &dccp_delack_timer,
                             &dccp_keepalive_timer);
}

static void tcp_write_timer(unsigned long);
static void tcp_delack_timer(unsigned long);
static void tcp_keepalive_timer (unsigned long data);

// net/ipv4/tcp_timer.c
void tcp_init_xmit_timers(struct sock *sk)
{
   inet_csk_init_xmit_timers(sk, &tcp_write_timer,
```

```
                                &tcp_delack_timer,
                                &tcp_keepalive_timer);
}
```

Both of these pass specific functions to `inet_csk_init_xmit_timers`, so after recognizing `dccp_write_timer`, `dccp_keepalive_timer`, and so forth as possible targets for the `fn(data)` call in `__run_timers`, there is no further propagation work to do.

4.2 Algorithm

The output of the function pointer analysis is a set of possible callees for the indirect calls of the analyzed program. We can store this information in a summary database `sum_funptr`.

```
session sum_funptr(FN:string,I:c_instr)
   containing [sindirect].
predicate sindirect(CFN:string).
```

The fact `sum_funptr(FN,I)->sindirect(CFN)` means that indirect call `I` within function `FN` might have `CFN` as a callee. `sum_funptr(FN,I)` might contain many such facts, and if the summary is conservative then together these facts describe a superset of all possible callees which might be targeted by this indirect call at runtime.

In order to get such a conservative approximation of the call graph, we will follow the propagation shown in the previous example. This propagation needs to keep track of which trace locations might flow to which indirect calls in the analyzed program. We can then follow these trace locations backward through assignments until we reach the original function assignment.

This information is stored with an intermediate summary database `sum_funptr_prop`, storing for each trace location the set of indirect calls which that location might flow to through assignments.

```
session sum_funptr_prop(L:t_trace_loc) containing [sref].
predicate sref(FN:string,I:c_instr).
```

We do not, however, consider arbitrary trace locations during propagation, and will restrict the possible entries in `sum_funptr_prop` both to constrain the search space and to avoid issues with aliasing (for propagation over arbitrary trace locations see Chapter 5). Specifically, the locations are restricted to either:

- `tl_func{_,drf{root{R}},cxt_any}`, where `R` is a function argument, local, or temporary variable.
- `tl_glob{drf{root{_}}}`, a global variable (typically a static array of function pointers).
- `tl_comp{C,drf{fld{empty,_,C}}}`, the value of a structure field.

Additionally, locations in `sum_funptr_prop` must have a static type identifying them as function pointers, e.g. ruling out variables and fields of type `void*` or `long`.

Now, the propagation algorithm is as follows:

1. For each indirect call `I` within a function `FN`, get a location `L` fitting our restrictions for the access path used to invoke the call; fail if there is no such `L`. Add `sref(FN,I)` to `sum_funptr_prop(L)`.
2. For each flow edge `L -> NL` (see Section 3.2.2) where `NL` matches some `NL'` and `L` is a function variable `CFN`, then for each `sref(FN,I)`

in `sum_funptr_prop(NL')`, add a predicate instance for the indirect call `sum_funptr(FN,I)->sindirect(CFN)`. This may introduce new flow edges for the indirect call site's argument and return variable bindings.

3. For each flow edge `L -> NL` where `NL` matches some `NL'` and `sum_funptr_prop(NL')` is not empty, get a location `L'` fitting our restrictions which matches `L`; fail if there is no such `L'` and `L` is not an actual function variable. Insert each predicate instance in `sum_funptr_prop(NL')` into `sum_funptr_prop(L)`.

4. For each flow edge `L -> NL`, if the dereference of `L` (`L` composed with `drf{empty}`) matches some `L'` and `sum_funptr_prop(L')` is not empty, then fail. This rules out cases where a variable or field storing a function pointer has its address taken.

5. As the entries in `sum_funptr_prop` grow, fixpoint the previous rules.

If the fixpoint is reached without failures, the final `sum_funptr` is a conservative approximation of the indirect call graph. If failures are encountered on any locations `L`, the call graph may be incomplete for any call sites `FN/I` such that `sum_funptr_prop(L)->sref(FN,I)` holds. In these cases we will fall back on manual annotation (see Section 9.2 for details of these annotations).

Chapter 5

Escape Analysis

A couple of symmetric questions (among many others) come up while perusing large code bases.

- When a value is read out of the heap and used, where did that value come from?
- When a value is written into the heap, where will that value eventually be used?

The escape analysis described here tries to answer these two questions with a fast, simple and, in the common case, accurate algorithm. We need a way to match up writes with their corresponding reads, and then transitively follow these assignment edges either forward or backward (flow-insensitively) through the program.

5.1 Example

Recall the `timer_list` problem discussed in Section 1.2. We are interested in where the `saa7146_buffer_timeout` function might be called,

after being assigned to the `function` field of a `timer_list`. One call site we would like to exclude is in `ctnetlink_del_conntrack`.

```
// 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;
}
```

While we were able to prove that `ctnetlink_del_conntrack` cannot call `saa7146_buffer_timeout` due to incompatibility between the structures the different `timer_list`'s are nested in, we are more generally interested in which functions `ctnetlink_del_conntrack` **can** call. If we can find a narrow set of targets for the call site, we can refine the results of the function pointer analysis accordingly, and will handle the comparison task not just for the `saa7146_buffer_timeout` `timer_list` but all other `timer_list`'s as well.

We can find a narrower set of targets for the `ct->timeout.function` call by following the function pointer backwards, in the same manner as the function pointer analysis but at a more precise level of granularity. Instead of considering all values of type `timer_list` and what the `function` field is assigned, we will consider all values of type `ip_conntrack` and what the `timeout.function` field is assigned.

There is only a single place in Linux where the `timeout.function` field is *directly* assigned, in `ip_conntrack_alloc`.

```
// net/ipv4/netfilter/ip_conntrack_core.c
static void death_by_timeout(unsigned long ul_conntrack)
{
   struct ip_conntrack *ct = (void *)ul_conntrack;
   ...
}

// net/ipv4/netfilter/ip_conntrack_core.c
struct ip_conntrack *ip_conntrack_alloc(...)
{
   struct ip_conntrack *conntrack;
   ...

   conntrack = kmem_cache_alloc(ip_conntrack_cachep, GFP_ATOMIC);
   ...

   init_timer(&conntrack->timeout);
   conntrack->timeout.data = (unsigned long)conntrack;
   conntrack->timeout.function = death_by_timeout;

   ...
}
```

From this can we conclude that `timeout.function` can only point to `death_by_timeout`? No. As seen here, other functions such as `init_timer` receive the address of `conntrack->timeout`, such that they have a `timer_list*` which could be used to write `timeout.function` indirectly. To be sure we have captured all writes to `timeout.function`, we have to consider the possibility of such indirect writes.

There are two strategies we can use to check for indirect writes. First, look at `init_timer` and all other points where `&timeout` is taken, and follow the value holding that address through the program, seeing if its `function` field is written anywhere. Second, look at all

writes to the `function` field of any `timer_list`, and figure out which of those `timer_list`'s might have come from the `timeout` field of an `ip_conntrack`.

These strategies correspond to two directions in a graph search, looking for paths from sources (points where `&timeout` is taken) to sinks (points where some `function` field is written). Which strategy will be more successful depends on the vagaries of the particular structure and code base examined; how much code initializes or uses the structure, and how complicated that code is. We want a mechanism for escape analysis that supports both, so that in subsequent analyses we can use either strategy as appropriate.

5.1.1 Addresses of the `timeout` field

Our first strategy is to figure out if there is any point where `&timeout` is taken, such that that address might be used to subsequently write to `timeout.function`. We have to find all the places where `&timeout` is taken and do a *forward* escape propagation to find all the places where that address might be used, and see what values `function` is written with.

Fortunately, the original point we noticed `&timeout` being taken, `init_timer`, is pretty straightforward, and does not write the `function` field nor pass the `timer_list` on to any other function or heap location.

```
// kernel/timer.c
void fastcall init_timer(struct timer_list *timer)
{
   timer->entry.next = NULL;
   timer->base = per_cpu(tvec_bases, raw_smp_processor_id());
}
```

Unfortunately, there are more complicated cases than `init_timer`.

__ip_conntrack_confirm might call add_timer with &timeout, which is the kernel function which actually schedules the timer_list for subsequent execution.

```
// net/ipv4/netfilter/ip_conntrack_core.c
int
__ip_conntrack_confirm(struct sk_buff **pskb)
{
   struct ip_conntrack *ct;
   ct = ip_conntrack_get(*pskb, &ctinfo);

   ...
   if (...) {
      ...
      ct->timeout.expires += jiffies;
      add_timer(&ct->timeout);
      ...
   }
   ...
}
```

add_timer will store the timer in the kernel data structures which manage pending timers, and will eventually be removed by __run_timers and run. The function and data fields will not be modified throughout the course of this, but we will need to show that in order to establish that death_by_timeout is the only possible target for some timeout.function field of an ip_conntrack.

```
// include/linux/timer.h
static inline void add_timer(struct timer_list *timer)
{
   BUG_ON(timer_pending(timer));
   __mod_timer(timer, timer->expires);
}

// kernel/timer.c
int __mod_timer(struct timer_list *timer, unsigned long expires)
```

```
{
   tvec_base_t *base;
   ...
   base = lock_timer_base(timer, &flags);

   ...
   timer->expires = expires;
   internal_add_timer(base, timer);
}

// kernel/timer.c
static void internal_add_timer(tvec_base_t *base,
                               struct timer_list *timer)
{
   unsigned long expires = timer->expires;
   unsigned long idx = expires - base->timer_jiffies;
   struct list_head *vec;

   if (idx < TVR_SIZE) {
      int i = expires & TVR_MASK;
      vec = base->tv1.vec + i;
   } else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
      int i = (expires >> TVR_BITS) & TVN_MASK;
      vec = base->tv2.vec + i;
   } else if (...) {
      ...
   }

   list_add_tail(&timer->entry, vec);
}
```

Calling `add_timer` will have the effect of placing the input `timer_list` onto a doubly-linked list in a `tvec_base_t` structure.

```
// kernel/timer.c

typedef struct tvec_s {
   struct list_head vec[TVN_SIZE];
} tvec_t;
```

```
typedef struct tvec_root_s {
   struct list_head vec[TVR_SIZE];
} tvec_root_t;

struct tvec_t_base_s {
   spinlock_t lock;
   struct timer_list *running_timer;
   unsigned long timer_jiffies;
   tvec_root_t tv1;
   tvec_t tv2;
   tvec_t tv3;
   tvec_t tv4;
   tvec_t tv5;
} ____cacheline_aligned_in_smp;

typedef struct tvec_t_base_s tvec_base_t;
```

The `tvec_base_t` structure is the core data structure used by the kernel to manage pending timers. This consists of five arrays of doubly-linked lists, at fields `tv1` through `tv5` of the `tvec_base_t`. These vectors were filled in by `internal_add_timer`, and will be read out by `__run_timers`.

```
// kernel/timer.c
static inline void __run_timers(tvec_base_t *base)
{
   struct timer_list *timer;

   while (time_after_eq(jiffies, base->timer_jiffies)) {
      struct list_head work_list = LIST_HEAD_INIT(work_list);
      struct list_head *head = &work_list;
      int index = base->timer_jiffies & TVR_MASK;

      ...
      list_splice_init(base->tv1.vec + index, &work_list);
      while (!list_empty(head)) {
         void (*fn)(unsigned long);
```

```
            unsigned long data;

            timer = list_entry(head->next,struct timer_list,entry);
            fn = timer->function;
            data = timer->data;

            set_running_timer(base, timer);
            detach_timer(timer, 1);

            ...
            fn(data);
            ...
        }
    }

    set_running_timer(base, NULL);
}
```

__run_timers splices lists of timers from the tvec_base_t into its local work_list, scans through those lists and runs the timers, without modifying the function or data fields. There are a few other similar functions which walk the tvec_base_t lists, and which also do not touch the function and data fields. We must be able to show that these are the only functions which might use the timer_list which was originally inserted via add_timer. In order to do this, we need some reasonably precise model of what these lists are doing.

5.1.2 Writes to the function field

It was a lot of work to follow that timeout field through the internal timer structures of the kernel, just to see if they led to any writes of the function field. Our second strategy will avoid this by looking at writes to the function field itself, and see which of those could have come from the timeout field. While there are nearly 600 places in Linux where

the `function` field is written, the vast majority of this is initialization code, and avoids the hairy lists and arrays used by `__run_timers`.

In fact, of all those writes to the `function` field, it is trivial to show that all but five of them cannot alias `timeout.function` — they directly write to the `function` field of a `timer_list` either embedded in a structure other than `ip_conntrack` (e.g. recall the `vbi_init` and `video_init` functions from Section 1.2), or to the `function` field of a statically allocated global `timer_list`.

One of the five remaining writes is `ip_conntrack_alloc` itself, which fills in `timeout.function` directly with `death_by_timeout`. The remaining four are utility functions which wrap the initialization of the `function` and `data` fields of a `timer_list`, such as `setup_timer`.

```
// include/linux/timer.h
static inline void setup_timer(struct timer_list * timer,
                               void (*function)(unsigned long),
                               unsigned long data)
{
   timer->function = function;
   timer->data = data;
   init_timer(timer);
}
```

Whenever `setup_timer` and the other three utility functions are called, they are passed the address of either some `timer_list` embedded in a structure, or the address of a statically allocated global `timer_list`. When we do a backward propagation from the first argument to `setup_timer`, we will find all these locations and be able to show they cannot alias the `timeout` field of an `ip_conntrack`.

```
void br_stp_port_timer_init(struct net_bridge_port *p)
{
   setup_timer(&p->message_age_timer,
```

```
                br_message_age_timer_expired,
                (unsigned long) p);

    setup_timer(&p->forward_delay_timer,
                br_forward_delay_timer_expired,
                (unsigned long) p);

    setup_timer(&p->hold_timer,
                br_hold_timer_expired,
                (unsigned long) p);
}
```

This second strategy requires more work than the first strategy to look at all the initializatinon code for a `timer_list`, but has the advantage in that the code involved is much simpler and easier to understand. However, this is not always the case and, again, in general we need to be able to follow values both backwards and forwards through the program.

5.2 Algorithm

The core of the escape analysis is just to compute a closure over the assignments in the program for a particular `t_trace_loc` – construct a set of `t_trace_loc`'s which collectively describe where the locations identified might flow. This information is computed in a demand-driven fashion by the `escape_set` predicate.

```
type t_escape_precision.

type t_escape_state ::=
    es_forward
  | es_backward
  | es_forward_precise{V:t_escape_precision}
  | es_backward_precise{V:t_escape_precision}.
```

```
predicate escape_set(in TR:int, in Y:t_escape_state,
                     in L:t_trace_loc,
                     out NLS:set[t_trace_loc])
    succeeds [zero,once].
```

`escape_state` takes an initial location `L`, state information `Y` indicating whether the search is forward or backward over assignments (and possibly some precision information on the search granularity, see Section 5.2.1), and an exploration threshold `TR`. `NLS` is bound to a set containing all the locations that `L` might flow to according to `Y`. `escape_set` may fail if the resulting set `NLS` is larger than the threshold `TR`, in which case the set of reachable locations is unknown.

`escape_set` performs a graph search that looks at one `t_trace_loc` at a time and identifies all the flow edges in the program that are relevant to that location and move its contents either forwards or backwards into another location.

However, as we are considering arbitrary trace locations we need to consider the indirect effects of aliasing on data flow. If we are tracking forward the location representing `x->f->g` within some function, and see an assignment `y = x->f->g`, we know the location has propagated forward to `y`. In addition, if we see another assignment `z = x->f`, we know the location has propagated forward to `z->g`. If we change these assignments around, so that we see `x->f->g = y`, we do **not** propagate to `y`, as we are propagating forward and are only interested in where `x->f->g` is used. The same does not hold for `x->f = z`: this assignment indicates `x->f` and `z` are potentially aliased and that the contents of `x->f->g` could propagate forward due to some assignment `a = z->g`, so we must propagate to `z->g`.

1. For a location `L`, enumerate all its sublocations `SL` and offset traces

`RT`.

2. For each such sublocation and offset trace, find all flow in the program either from a location matching `SL` into a new location `XSL`, or from another location `XSL` into a location matching `SL`, according to whether this is a forward or backward propagation. If `RT` includes one or more dereferences (`drf{_}`), consider **both** forward and backward assignments.

3. Compose `XSL` with `RT` to get the new location `XL` to which the location at `L` is propagated.

4. Generalize `XL` if necessary, simplifying it to describe a superset of access paths, and goto 1.

Generalizing trace locations allows the escape analysis to adjust its precision in order to do potentially less work for a potential small difference in the final results of the analysis. If location `XL` represents `x->f->g` where the type of `x` is `str1*`, we could generalize to location `.f->g` relative to values of type `str1`. Doing so means we would no longer have to consider assignments over `x` itself, but would now have to look at uses of all values of type `str1`. Similar trade-off's exist for other generalization possibilities, either to `.g` relative to whatever the type of `x->f` is, or just leaving the trace `x->f->g` intact.

The choice of generalization heuristic can have a very large effect on the result of `escape_set`. Over-generalizing can degrade precision to the point of uselessness to the client, while under-generalizing can expand the search space to the point of intractability. The right balance depends ultimately on the client's own property and which parts of the trace are actually relevant to that property. However, all the clients we have used have been covered by one of several simple techniques.

5.2.1 Trace Generalization

Going back to the `t_escape_state` definition:

```
type t_escape_state ::=
    es_forward
  | es_backward
  | es_forward_precise{V:t_escape_precision}
  | es_backward_precise{V:t_escape_precision}.
```

The state here encodes, alongside the forward/backward direction, all information about how much generalization `escape_set` should perform during the location space search. `es_forward` and `es_backward` apply the maximal generalization: for locations that reference fields, `tl_comp` is used at the outermost field access, while, for locations without fields, `tl_glob` is used for global variables and all context information is removed from arguments and locals to a `tl_func`.

`es_forward_precise` and `es_backward_precise` use a more precise generalization, according to the `t_escape_precision` value used.

```
type t_escape_precision ::=
    ep_field{FLD:int}
  | ep_nocontext
  | ep_maximum.
```

Level `ep_field{N}` generalizes to `tl_comp` only for locations where more than `N` fields are used in the trace access path, preserving the outermost `N` fields, e.g. `ep_max_field{2}` will generalize `x->f.g.h` to `tl_comp{str,.g.h}` where `str` is the type of `x->f`. Level `ep_nocontext` never generalizes to `tl_comp` but strips all context information from `tl_func` locations to reduce the search space at the expense of considering non-CFL-feasible paths in the call graph (exploring into a function

via one call site and then back out at a different call site to the same function). Level `ep_maximum` never generalizes at all.

Even with this variety of generalization heuristics, there is still rarely a single best option for a given client analysis. A strength of this analysis is that the individual `escape_set` queries are independent of one another, and in trickier cases a client can try to escape using multiple different heuristics. If `escape_set` exploration for any of the heuristics succeeds, the resulting set of locations is an overapproximation of the places the original location could flow to. If multiple heuristics succeed, we can get the best bound as the set of traces which are described by **all** the succeeding heuristics. See Section 6.5.2 for an instance of this approach.

5.2.2 Handling Arrays and Recursive Structures

Arrays and recursion can introduce an unbounded number of access paths which the escape analysis might need to consider. In the context of our earlier example, for `internal_add_timer` we may need to consider any array element of `base->tv1.vec`, and any element of the lists stored in this array. The `t_trace` type as presented so far does not give us the expressiveness for this; for arrays we would consider `base->tv1.vec[0]`, `base->tv1.vec[1]`, and so forth, and for lists we would consider `base->tv1.vec[0]`, `base->tv1.vec[0]->next`, `base->tv1.vec[0]->next->next`, and so forth.

So we define new traces which represent elements chosen arbitrarily from arrays and lists.

```
type t_trace ::=
  ...
  | uk_index{T:t_trace,Y:t_type}
  | uk_recurse{T:t_trace,RL:list[t_trace]}
```

.

Trace `uk_index{T,Y}` represents any arbitrarily chosen `index{T,Y,_}` trace, and `uk_recurse{T,RL}` represents any arbitrarily chosen trace reachable from `T` (excluding `T` itself) over the recursive structure backbone given by `RL` — for the `list_head` structure, which is a doubly-linked list with `prev` and `next` members, `RL` is the set:

```
[drf{fld{empty,"prev","list_head"}},
 drf{fld{empty,"next","list_head"}}]
```

Returning to our earlier example, we can use `uk_recurse` to model the list insertions and manipulations performed by the `list_add_tail` and `list_splice_init` called by `internal_add_timer` and `__run_timers`.

```
// include/linux/list.h
static inline void list_add_tail(struct list_head *new,
                                 struct list_head *head)
{
   // inlined call __list_add(new, head->prev, head);
   struct list_head *prev = head->prev;
   head->prev = new;
   new->next = head;
   new->prev = prev;
   prev->next = new;
}

static inline void list_splice_init(struct list_head *list,
                                    struct list_head *head)
{
   if (!list_empty(list)) {
      // inlined call __list_splice(list, head);
      struct list_head *first = list->next;
      struct list_head *last = list->prev;
      struct list_head *at = head->next;

      first->prev = head;
```

```
        head->next = first;

        last->next = at;
        at->prev = last;

        INIT_LIST_HEAD(list);
    }
}
```

`list_add_tail` assigns `new` to `head->prev`, which we can generalize to `uk_recurse{*head}`, so that the new element becomes threaded onto the list pointed to by `head`. Similarly, `list_splice_init` assigns `list->next` to `head->next`; since `list->next` is matched by `uk_recurse{*list}`, and `head->next` generalizes to `uk_recurse{*head}`, members of `list` become threaded onto the list pointed to by `head`.

Filling out our earlier example with timer usage, we can now follow the complete propagation of the `timer_list` parameter to `add_timer` until it is read back out by `__run_timers`. The following is a single path to `__run_timers`; there are several other similar paths.

1. `tl_func{"add_timer",*timer,cxt_any}`

   Initial location, the first argument to `add_timer`.

2. `tl_func{"__mod_timer",*timer,cxt_call{"add_timer",cxt_any}}`
   `-> tl_func{"__mod_timer",*timer,cxt_any}`

   Passed as argument to `__mod_timer`. The trace location is generalized to remove the calling context.

3. ```
   tl_func{"internal_add_timer",*timer,
           cxt_call{"__mod_timer",cxt_any}}
     -> tl_func{"internal_add_timer",*timer,cxt_any}
   ```

   Passed as argument to `internal_add_timer`. The trace location is generalized to remove the calling context.
   ```

4. `tl_func{"list_add_tail",(*new)^entry,`
   `cxt_call{"internal_add_timer",cxt_any}}`

   Internal `entry` field of the timer is passed to `list_add_tail`.

5. `tl_func{"list_add_tail",(*next->prev)^entry,`
   `cxt_call{"internal_add_timer",cxt_any}}`
   `-> tl_func{"list_add_tail",uk_recurse{*next}^entry,`
   `cxt_call{"internal_add_timer",cxt_any}}`

   `list_add_tail` attaches the timer's `entry` to its `next` parameter (the pointer to the list tail). This is generalized with `uk_recurse` so that the `entry` is now reachable via any chain of `next`/`prev` pointers from the `next` parameter.

6. `tl_func{"internal_add_timer",uk_recurse{*vec}^entry,cxt_any}`

   Propagated back out to `internal_add_timer`, using the context on the `list_add_tail` trace location.

7. `tl_func{"internal_add_timer",`
   `uk_recurse{base->tv1.vec[i]}^entry,cxt_any}`
   `-> tl_func{"internal_add_timer",`
   `uk_recurse{uk_index{base->tv1.vec}}^entry,cxt_any}`
   `-> tl_comp{"tvec_base_t",`
   `uk_recurse{uk_index{.tv1.vec}}^entry}`

   Follow assignment to the local variable `vec` from the `tv1.vec` buffer in the input `base` parameter. The trace location is generalized to the type of `base`.

8. `tl_func{"list_splice_init",uk_recurse{*list}^entry,`
   `cxt_call{"__run_timers",cxt_any}}`

   `tv1.vec` element is passed as argument to `list_splice_init` by `__run_timers`.

9. `tl_func{"list_splice_init",(*first)^entry,`
   `cxt_call{"__run_timers",cxt_any}}`

`next` pointer of the `list` is copied to the local variable `first`.

10. ```
    tl_func{"list_splice_init",(*head->next)^entry,
            cxt_call{"__run_timers",cxt_any}}
      -> tl_func{"list_splice_init",uk_recurse{*head}^entry,
                 cxt_call{"__run_timers",cxt_any}}
    ```

    `first` is copied to the `next` pointer of the parameter list `head`. This is generalized to reintroduce the `uk_recurse` trace.

11. ```
    tl_func{"__run_timers",uk_recurse{work_list}^entry,cxt_any}
    ```

    Propagate back out to caller `__run_timers`.

12. ```
    tl_func{"__run_timers",uk_recurse{*head}^entry,cxt_any}
    ```

    Follow assignment of `&work_list` to local variable `head`.

13. ```
    tl_func{"__run_timers",*timer,cxt_any}
    ```

    Use `container_of` to access the base timer of the `head->next` list entry, completing the propagation.

5.2.3 Cacheing and Optimization

We can use some simple analysis to greatly improve the escape analysis performance on many queries. The base escape analysis hence described is completely demand driven, with no cacheing between queries to allow reuse of the exploration of overlapping parts of the state space.

We can get part of the effect of a cacheing mechanism, however, with simple, specialized analyses that cover many of the most expensive and frequently repeated portions of the state space. These analyses can, in many cases, make the difference between tractability and intractability for future queries, by greatly reducing the number of locations those queries will have to visit.

Viable uses of `rfld`

Suppose we are tracking backward `*(x->g)` and see an assignment `y = &x->f`. This introduces the implicit flow edge from `*x` to `rfld{*y,f,c}` (See Section 3.2.2), and since `*x` is a sublocation of the location we are tracking, we have to follow the edge and consider `*(rfld{*y,f,c}.g)`. If there is no place in the program where `container_of` is ever used with field `f`, the location to which we propagated will never be accessed anywhere in the program, and any further propagation we do is meaningless.

We thus perform an analysis to find all fields which are used with `container_of`, and for fields `f` **not** in this set, ignore flow edges with `rfld{_,f,c}` in the target.

Read only data passed to functions

Suppose we are, again, tracking backward `*(x->g)` and see a call `foo(x);`. Again, we have to follow `*x` to the new location `*(y->g)` where `y` is the first argument to `foo`, and following that to everywhere `y` is passed, and so forth. For any of the locations where `y` flows to, we are only interested in what new values could be assigned to `y->g`. Many of the arguments passed to functions will only be used for reading, and in these cases following the argument edge forward during backward propagation will not yield anything.

To avoid this useless propagation, we perform an analysis to find argument traces to each function which are read-only: they never escape forward to a point where they are written. (Note that we consider all writes, even those outside the function, for cases where the location escapes somewhere within the function, and is then written elsewhere in the program). This is just a scan over each function in the program,

applying the escape analysis to the fields of each structure pointer argument (following nested structures transitively). Traces are marked readonly in cases where the escape succeeds and none of the resulting locations are ever written to.

Relative field writes

Suppose we are tracking backward `*(x->f.g)` and see an assignment `y = &x->f`. Again, we now have to follow `*(y->g)` to see if there are assignments to the `g` field we need to continue following back. In some cases `y` could flow forward to many hundreds of locations, and yet none of those will possibly result in a write since all writes to `g` are performed *relative* to an outer structure, of the form `z->f.g = ...` or `q.g = ...` with `q` a global variable. The assignment to `y` could only lead to a write to `g` in places where some `z->f.g` is written, even if we have no idea where `y` flows.

The `timer_list` discussed in Section 1.2 is an example of this; all `timer_list`'s are either part of a larger structure or are global variables, and that larger structure or variable is used when writing the `func` and `data` fields of the `timer_list`.

We can handle this case with yet another scanning analysis. For each field `g` in the program, we will mark writes `y->g = ...` as non-relative, and writes `z->f.g = ...` or `q.g = ...` as relative. When escaping, for fields `g` with no non-relative writes, taking the address `y = &x->f` will lead to a future write only at those relative writes where some `z->f` is specified.

5.3 Memory Analysis Refinements

With the escape analysis in hand, we can now revisit the question of generating sound rules for the local memory analysis on when traces might alias, and when they might be clobbered by a function call. These correspond to filling in rules for the predicates `trace_alias` and `trace_clobber` (Section 3.3.5), and are covered in Sections 5.3.1 and 5.3.2, respectively.

5.3.1 Local Memory Aliasing

The only aliasing queries generated by the local memory analysis will be for those traces which can hold a value, i.e. are of integral or pointer type and are not the base of a structure. This includes `root`, `drf`, `fld`, and `index` traces (`rfld` traces are always the base of a structure).

Syntactically identical traces always alias one another; the location referred to by a trace stays the same throughout the execution of a function. For two syntactically different traces, the following conditions describe when aliasing might occur.

- Two traces `root{_}` and `root{_}` never alias.
- Two traces `root{_}` and `fld{_,_,_}` never alias.
- Two traces `fld{T0,F0,C0}` and `fld{T1,F1,C1}` alias iff `F0 = F1`, `C0 = C1`, and `T0` and `T1` alias. This assumes the type safety property from Chapter 1, that if the two fields are different they cannot overlay.
- Two traces which are both either `drf` or `index` alias if they have the same type. This assumes a similar notion to type safety, that

two locations which are **not** fields of a structure are used with a consistent type, e.g. no location is used separately as an integer and a pointer.

- Two traces `index{_,_,_}` and either a `root` or `fld` never alias.
- Two traces `drf{_}` and either a `root` or `fld` do not alias if they are different types. If they are the same type, they might alias **unless** the escape analysis shows that either the `root/fld` cannot flow forward to the `drf`, or that the `drf` cannot flow backward to the `root/fld`.

The last rule is the interesting one. For a variable `x` and pointer `y`, `x` and `*y` can alias only if somewhere in the program `&x` is taken and there is a path through the assignment flow edges into `*y`. We can disprove the existence of such a path by finding either all the points where `x` flows to, or where `*y` comes from, and showing the other is not in that set.

5.3.2 Local Memory Clobbering

Constructing precise per-function usemod information is an extremely difficult problem, and one that we are not going to try to tackle. The version of the Linux kernel we are analyzing contains 91384 functions. The call graph produced after running our function pointer analysis includes a strongly connected component of 48415 functions, where an additional 20157 functions are called by functions in the SCC, and 16357 functions call into functions in the SCC (the remaining 6455 functions are disconnected from the SCC). According to this call graph, then, 70% of the functions in the kernel can call into 75% or more of

the functions in the kernel, and could transitively modify all the data written by those functions.

The call graph precision could be improved, of course, with a better function pointer analysis that generates more precise and context-sensitive call information. For example, consider the utility function `kref_put`, which decrements reference counts on kernel objects:

```
// lib/kref.c
int kref_put(struct kref *kref,
             void (*release)(struct kref *kref))
{
   WARN_ON(release == NULL);
   WARN_ON(release == (void (*)(struct kref *))kfree);

   ...
   if ((atomic_read(&kref->refcount) == 1) ||
      (atomic_dec_and_test(&kref->refcount))) {
         release(kref);
         return 1;
   }
   return 0;
}
```

`kref_put` is called in dozens of places by unrelated parts of the kernel, with the `release` callback leading back into those parts. A context-insensitive call graph will conflate all the parts of the kernel which use `kref_put` by drawing paths from each caller to all the possible callbacks. A context-sensitive call graph, by splitting the call to `release` according to who called `kref_put`, will generate paths from each caller only to the corresponding `release` callback.

We've tried this improvement, with k-limited ($k = 1$ usually, or more for some functions) context-sensitive indirect call edges (this is done using the escape analysis on trace locations containing the indirect

call's function pointer trace and `cxt_call` contexts for each parent call site), to little effect in the size of the call graph SCCs.

It may well be, then, that this huge SCC and the huge list of callees and modsets for most functions reflects the real state of the kernel. We can get some leverage on the modset problem, however, by handling two common cases that will be important to later analyses: structure fields which are only modified at creation of the structure, and functions which modify few or no locations.

We'll call these *semi-pure* fields and functions. A semi-pure field is one where only a small fixed set of initialization functions can transitively modify that field in a structure passed into the function (via arguments or global variables). A semi-pure function is one which can transitively modify only a small fixed set of the locations passed into it. Inference for field and function purity is covered in Sections 5.3.2 and 5.3.2, respectively, and the call site clobbering rules incorporating these is shown in Section 5.3.2.

Structure field purity

Semi-pure fields are only written either at creation of the parent structure or very shortly afterwards by initialization functions. These can be viewed as something of a backbone for the heap, important fields stabilizing the heap's structure by creating links between structures which persist until the structures are deallocated. To capture these we will use the `salias_comp_init` session:

```
session salias_write_comp(C:string,F:string)
    containing [wcomp_impure,wcomp_init].
predicate wcomp_impure(FN:string).
predicate wcomp_init(FN:string,T:t_trace).
```

These predicates have the following interpretation:

- `salias_write_comp(C,F)->wcomp_impure(FN)`: There is a write to `F` in `FN` that is not initialization code. The structure being written was passed in from outside and could be any structure of type `C` in the heap.

- `salias_write_comp(C,F)->wcomp_init(FN,T)`: There is a write to field `F` of `T` in `FN` or one of its transitive callees that is initialization code. All functions to which `T` was passed from the point it was allocated to the point of the write have a `wcomp_init` predicate in their summary (this is a small set of functions).

A field with no `wcomp_impure` predicates in its `salias_write_comp` session is semi-pure. A trace `fld{T,F,C}` where `F` is semi-pure can be modified by a call to a function `CFN` only if there is a `wcomp_init(CFN,CT)` predicate and `CT` within `CFN` can alias `T`.

We can infer these predicates with a scan over the code base. For any write to some `fld{T,F,C}`, use the escape analysis to follow the base structure backward at the `ep_maximum` level of precision, so that we never generalize the trace. This escape will either terminate at all the possible allocation sites for `T` (stack locals, globals, or return values of `kmalloc` and other primitive allocators), or it will spin off into the heap and fail to capture the possible sources of `T`. We mark the write as `wcomp_impure` if the escape fails, or if it succeeds but the sources include any traces rooted at global variables. Otherwise, we take every `tl_func{FN,T,_}` from the result of the escape, where `T` is rooted at an argument, and add `wcomp_init(FN,T)`.

Note that this procedure will not generate **any** summary information for writes to structure fields within the function the structure is allocated (e.g. `str *x = (str*)kmalloc(sizeof str); x->f = ...;`). Since the data being written was allocated inside the function

these writes are not part of the function's modifies set.

Function purity

Of the 26612 functions in the analyzed Linux kernel which do not call into the large SCC, 11768 (44%) do not have any callees at all, and 24745 (93%) have fewer than 10 transitive callees. For most of these functions we can construct an exact modifies set. We use the `salias_write_func` session to store per-function modset information:

```
session salias_write_func(FN:string)
    containing [wfunc_impure,wfunc_write,
                wfunc_write_inline,wfunc_allocate].
predicate wfunc_impure().
predicate wfunc_write(T:t_trace).
predicate wfunc_write_inline(T:t_trace,V:scalar).
predicate wfunc_allocate(T:t_trace).
```

These predicates have the following interpretation:

- `salias_write_func(FN)->wfunc_impure()`: Nothing is known about the precise set of locations `FN` might modify while executing.
- `salias_write_func(FN)->wfunc_write(T)`: Trace `T` as passed into `FN` might be written to.
- `salias_write_func(FN)->wfunc_write_inline(T,V)`: Trace `T` as passed into `FN` will definitely be written to with the value `V`, a `scalar` integer or trace value expressed in terms of the entry state to `FN`.
- `salias_write_func(FN)->wfunc_allocate(T)`: Trace `T` as passed into `FN` will definitely be written to either with `NULL` or with a freshly allocated location.

A function FN with no wfunc_impure in its salias_write_func session is semi-pure. In this case the wfunc_write predicates describe all the locations which might be written to by FN but were not allocated within the body of FN. Predicate wfunc_allocate characterizes wrappers for primitive allocation functions, and wfunc_write_inline is primarily for the numerous inline functions in Linux that perform some simple side-effect free integer operations and/or heap accesses before returning (these functions are usually used in lieu of macros). An example is netdev_priv, a utility function called over 3000 times in Linux which returns the address of a block of data hidden at the end of every net_device structure (we will take a closer look at this structure in Section 9.6.1).

```
// include/linux/netdevice.h
#define NETDEV_ALIGN        32
#define NETDEV_ALIGN_CONST  (NETDEV_ALIGN - 1)

static inline void *netdev_priv(struct net_device *dev)
{
   return (char *)dev + ((sizeof(struct net_device)
                                + NETDEV_ALIGN_CONST)
                         & ~NETDEV_ALIGN_CONST);
}
```

Inferring the wfunc_write_inline and wfunc_allocate predicates requires deeper reasoning about integers and must-update facts than what we can get with the global memory model used by the escape analysis. To get the predicates in salias_write_func we will switch over to the local memory model, which itself depends on salias_write_func for call site clobbering information (Section 5.3.2). It is safe to put the salias_write_func analysis in a fixpoint, though; initially the sessions will be empty indicating there are no side effects on a function, but

as the sessions get filled in any affected callers will be automatically reanalyzed using the more conservative information.

For each function we will first mark it as impure if any of its callees is impure. Then, for each write in the function, including both explicit writes and `wfunc_write` predicates propagated up from its callees:

1. Drop all writes to traces allocated within the function — local variables, freshly heap-allocated locations, and their fields.

2. Mark the function as impure if there are any remaining writes to traces containing `uc_sum{_,_,_}`. The `uc_sum` traces generated when pointers might be clobbered by a call are unconstrained and could point anywhere in the heap.

3. Apply an arbitrary cleanliness filter to the remaining writes, and mark the function as impure if any of them are filtered out. Writes we filter out include those to traces rooted at global variables, traces with more than two `drf` dereferences, and traces containing an `index` or `rfld`. These filters keep a lot of crud out of the summary sessions for impure functions, and also allow the analysis to terminate on functions that walk and modify the contents of arrays or recursive structures.

4. All the remaining writes are added to the `salias_write_func` session as `wfunc_write` predicates.

5. If any of the written traces always points to `NULL` or a freshly allocated heap location at function exit, add it as `wfunc_allocate`.

6. If any of the written traces always points to a particular value which doesn't use any `uc_sum{_,_,_}` unconstrained trace, add it as `wfunc_write_inline`. We also use a filter here, only adding

inline assignments for the return value and for traces which are assigned an argument of the function (a very common pattern in initialization functions).

Clobbering Rules

Once the `salias_write_comp` and `salias_write_func` databases have been generated, the rules for the `trace_clobber` are as follows. For calls to semi-pure functions, the only traces clobbered are those which might alias (via `trace_alias`) any of the `wfunc_write` predicates in the function's summary. For calls to impure functions, any trace might be clobbered except those meeting one of the following criteria:

- Local variables and their fields which do not escape forward into the called function or into any global variables.
- Fields `fld{T,F,C}` which are semi-pure and where there is not a `wcomp_init` predicate for `F/C` on the called function.

Additionally, at calls to a function containing a `wfunc_write_inline` predicate (whether the function is semi-pure or not), we can inline these writes at the call site by adding instances of the `assign` predicate at the call site, as if it were a regular assignment. These will be picked up by the local memory model and used to apply a strong update which does not sacrifice any precision by introducing a `uc_sum` trace.

Chapter 6

Polymorphic Data

Polymorphism is used extensively in the Linux kernel. Many of the core kernel data structures associated with filesystems, memory maps, devices, drivers, and almost all other kernel subsystems include some element of parametric or subtype polymorphism — a combination of function pointers and `void*` pointers used to extend a structure's functionality or the data it stores according to a particular interface.

Polymorphism in Linux typically follows one of two patterns, with examples of each in Sections 6.1 and 6.2. First, and simplest, is to store in a structure a function pointer and some `void*` data, such that at some point in the future the function pointer will be called with that data. For example, the timer mechanism (see Section 1.2) and interrupt handling mechanism use this approach. A function pointer and `void*` pointer are supplied, and at some point in the future, either when the timer expires or a particular interrupt occurs, the function is called with the data.

The second pattern is to use a whole table of function pointers, in essence a vtable in the object-oriented sense, which can write and read to a `void*` pointer with which a pointer to the table is correlated. The

`file` structure, an important filesystem component, includes a pointer to a table of function pointers for opening and closing the file, among numerous other operations. The function pointers in this table are used exclusively to access the `void*` field `private_data` of the associated `file`.

The analysis problem presented by these two patterns is the same; to follow what's happening in the code we need to precisely track the correlation between function pointers and targets of `void*` pointers.

6.1 Function Pointer Correlation Example

Let's revisit our very first example. Function `saa7146_buffer_timeout` might be called indirectly via `__run_timers` due to its use in `vbi_init` (and `video_init`).

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

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

// kernel/timer.c
```

```
static inline void __run_timers(tvec_base_t *base)
{
   struct timer_list *timer;
   ...

   void (*fn)(unsigned long);
   unsigned long data;

   timer = list_entry(head->next,struct timer_list,entry);
   fn = timer->function;
   data = timer->data;

   ...
   fn(data);
   ...
}
```

The reason the cast performed by `saa7146_buffer_timeout` is correct is that the `data` passed into it must be the same as that which was assigned by `vbi_init` — there is a correlation between the `function` and `data` fields of a `timer_list`. While the `function` and `data` fields are highly polymorphic and are assigned hundreds of different unrelated values within Linux, each such function is only associated with a very few possibilities for the data, and vice versa. Capturing these correlations and identifying what are the possible values of the data for each particular function is the purpose of the polymorphic data analysis. Polymorphic structures such as `timer_list` are in widespread use within Linux, and finding a near-exact model of these correlations is crucial to verifying type safety.

6.2 Function Pointer Table Example

Recall the saa7146 driver which contained the `saa7146_buffer_timeout` function used within `timer_list`. Here is another function in the saa7146 driver, `fops_read`.

```
// 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*`, so `fops_read` performs another 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 can interact with many devices as if they were regular files. To this end, there is a common interface within Linux for defining new files, which is neatly packaged up into the `file_operations` structure, basically a big table of function pointers, 27 in all (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 the `file` is primarily done through the function pointers in the `f_op` field pointing to the `file_operations` for the file. e.g. `vfs_read` is used to read out of a file as follows:

```
// 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;
}
```

`vfs_read` is called directly by the top level `sys_read` which is the

entry point from user space. To allow `vfs_read` and the other top level file operations to interact with saa7146 devices, the saa7146 driver creates a `file_operations` structure, and the `fops_read` function is set to the `read` pointer of this file. `vfs_read` may 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` might be invoked, why is the cast it performs correct? `video_fops` is never directly assigned to some `file->f_op`, and writes to the `f_op` field are never directly correlated with writes to the `private_data` field.

A different mechanism is in use. Function `__dentry_open`, which opens a file, sets the `f_op` field and proceeds to call 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;
   }

   ...
}
```

Going back to the `video_fops` used to store `fops_read`, we see that 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 polydata 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.

6.3 Algorithm Overview

When `saa7146_buffer_timeout` is called by `__run_timers`, the data supplied to it **must** be that specified by a previous call to `vbi_init` or `video_init`, not any of the hundreds of other assignments to the `data` field of a `timer_list` that exist in the Linux kernel.

The reasons for this are twofold. First, the `__run_timers` indirect call exploits a *structural relationship* between the function pointer target of the call and the first argument to the call: both of these are fields of the same `timer_list` structure. In general, a structural relationship is a pair of two 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 bear *structural correlations* with one another: the `function` field of a `timer_list` will be `saa7146_buffer_timeout` if and only if the `data` field was set by `vbi_init` or `video_init`, and similarly for the hundreds of other functions that may be used in a `timer_list`.

Structural relationships and correlations are a sufficiently general model to tackle our polymorphism problem, the algorithm for which we break into two phases. First, we scan all indirect call sites to identify 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 6.4). 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 6.5).

Both of these phases are safe but approximate. In the first phase

any relationship we find must indeed hold for that call site, but we may miss some interesting relationships that we cannot determine hold. In the second phase the correlations found will cover all possible values each function may be associated with, but may overapproximate or fail outright (in which case we know nothing about the correlations within the relationship).

6.4 Call Site Structural Relationships

For each indirect call site, we need a set of structural relationships which are guaranteed to hold at the call site. The output of this phase is stored in the `spoly_call` session:

```
session spoly_call(FN:string,I:c_instr)
    containing [wpoly_call_target,
                wpoly_call_comp,wpoly_call_func].

predicate wpoly_call_target(FT:t_trace).
predicate wpoly_call_comp(FT:t_trace,C:string,
                          RFT:t_trace,RT:t_trace,AT:t_trace).
predicate wpoly_call_func(FT:t_trace,T:t_trace,AT:t_trace).
```

Predicate `wpoly_call_target` identifies the possible traces `FT` for the function pointer used to invoke the indirect call site `I` within function `FN`. There may be multiple values for `FT` in cases where the indirect call targets different traces along different paths, such as in the `__dentry_open` example shown earlier, where the function pointer could either be the `open` function argument or the field `f->f_op->open`. Case splitting based on the function pointer's trace `FT` lets us identify the relationships holding when `f->f_op->open` is used irrespective of the case where `open` is used (there are only two possible `open` functions which might be passed in explicitly, both of which are no-ops).

For each of these traces FT, wpoly_call_comp and wpoly_call_func identify structural relationships which always hold on paths where FT is used to invoke the call. wpoly_call_comp identifies a relationship between two traces RFT and RT relative to a base structure of type C, where RFT is the function pointer invoking the call and RT is accessed through argument trace AT within the call. Similarly, wpoly_call_func identifies a relationship between two traces FT and T which are arguments to function FN, where FT is the function pointer invoking the call and T is accessed through argument trace AT within the call.

For the indirect call in the __run_timers function, there is a single structural relationship for the timer_list struct and the following predicates will be generated (we substitute the trace TIMERT for the fairly complicated trace which the variable timer is assigned):

```
wpoly_call_target(drf{fld{TIMERT,"function","timer_list"}}).
wpoly_call_comp(drf{fld{TIMERT,"function","timer_list"}},
                "timer_list",
                drf{fld{empty,"function","timer_list"}},
                drf{fld{empty,"data","timer_list"}},
                drf{root{arg{0}}}).
```

Computing the information in spoly_call is fairly straightforward and can be done without interprocedural analysis. Each indirect call is associated with a function expression FE which we apply the eval predicate to, yielding each function pointer trace FT and an associated guard FG. Then, for any trace T which might be passed in as argument trace AT if guard TG holds, we can add structural relationships between FT and T if FG implies TG.

The only issue is how to find the argument traces AT to check; while using just the plain argument drf{root{arg{_}}} is sufficient for the __run_timers example, the vfs_read example is concerned not with

the `file` argument but with its `private_data` field. More complex examples require even longer access chains, and in the general case the amount of data reachable from each argument (and thus the number of possible structural relationships) is unbounded.

We get around this issue with a mixture of heuristics and more robust methods. We will 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 downstream casting analysis (Chapter 7). The argument traces `AT` we will consider are:

- Untyped call arguments.
- Untyped fields of call arguments (or fields of fields, transitively, without following dereferences).
- Any additional trace specified via an auxiliary `spoly_call_try` session. This session specifies, for each function `XFN`, a set of traces `XAT` interesting to a downstream analysis. For the indirect call, we will consider each `XAT` associated with each potential target `XFN` of the call.

  The casting analysis will fill in `spoly_call_try` with all the untyped traces each function directly casts to a new type (e.g. for `saa7146_buffer_timeout` this will include `drf{root{arg{0}}}`).

6.5 Finding Relationship Correlations

The output of this phase is stored in the `spoly_receive_comp` and `spoly_receive_func` sessions:

```
session spoly_receive_comp(C:string,
                           RFT:t_trace,RT:t_trace,AT:t_trace)
    containing [wpoly_receive,wpoly_fail].

session spoly_receive_func(FN:string,
                           FT:t_trace,T:t_trace,AT:t_trace)
    containing [wpoly_receive,wpoly_fail].

predicate wpoly_receive(FNPTR:string,FN:string,DT:t_trace).
predicate wpoly_fail().
```

Each `wpoly_receive` predicate identifies a possible correlation for the structural relationship identified by the `spoly_receive_comp` or `spoly_receive_func` session which contains that predicate. If function `FNPTR` is the value held by the function pointer trace in the relationship (either `RFT` relative to type `C`, or trace `FT` within function `FN`), then the value held by the other trace `RT`/`T` may be that of `DT` within `FN`.

Unless a `wpoly_fail` predicate was generated for the relationship, then the `wpoly_receive` predicates collectively capture the possible values each concrete `FNPTR` may be correlated with.

The relationship for the `timer_list` struct will be stored in the following session:

```
spoly_receive_comp("timer_list",
                   drf{fld{empty,"function","timer_list"}},
                   drf{fld{empty,"data","timer_list"}},
                   drf{root{arg{0}}})
```

Due to the assignments in `vbi_init` and `video_init`, this session will contain the following predicates:

```
wpoly_receive("saa7146_buffer_timeout","vbi_init",
              fld{drf{root{arg{1}}},"vbi_q","saa7146_vv"}).
wpoly_receive("saa7146_buffer_timeout","video_init",
              fld{drf{root{arg{1}}},"video_q","saa7146_vv"}).
```

There will be hundreds of other `wpoly_receive` predicates in the same `spoly_receive_comp` session for `timer_list`, but no others will use `saa7146_buffer_timeout` for the function pointer.

We compute the `wpoly_receive` information by scanning all the writes in the program to determine:

1. Which structural relationships the write could affect — the heap location written may alias either the function pointer or value portion of the relationship or one of their subtraces (Section 6.5.1).

2. Which correlations the write may introduce for each relationship (Section 6.5.2).

These are written as two pieces of a single analysis (based on the local memory model from Section 3.3), and may both query and update the `spoly_receive_comp` and `spoly_receive_func` sessions. This will require an interprocedural fixpoint computation.

6.5.1 Writes Affecting Relationships

The output of this phase consists of the following two predicates, which collectively model the effect all the writes in the currently analyzed function have on the structural relationships in the `spoly_call` session.

```
predicate comp_assign_value(C:string,
                        RFT:t_trace,RT:t_trace,AT:t_trace,
                        NFT:t_trace,NT:t_trace).

predicate func_assign_value(FN:string,
                        FT:t_trace,T:t_trace,AT:t_trace,
                        NFT:t_trace,NT:t_trace).
```

`NFT` and `NT` are traces within this function for a new correlation for the relationship on either a `struct` type or function. Note that `NFT`

may not be a particular function; in Section 6.5.2 we will determine the possible functions it could refer to.

In the function case, the writes we are interested in are the argument bindings for calls to the target function. For any call to some function with a relationship on it, evaluate the relationship's argument traces `FT` and `T` at the call site (using `inst_trace`, Section 3.3.1) to get the new values `NFT` and `NT`.

The `struct` type case is considerably more complicated. Instead of argument bindings, we need to look at writes to the heap which could possibly affect the traces in each relationship. To help in analyzing the heap writes, we will make a couple of simplifying assumptions about the possible structural relationships. These assumptions are reinforced through filters on the `wpoly_call_comp` and `wpoly_call_func` predicates in the `spoly_call` session (Section 6.4).

- The only traces dereferenced in the paths on a relationship are structure fields (i.e. `drf{fld{_,_,_}}`). This rules out traces such as `(**x)->f[3]` (which in practice are not used in any meaningful relationships).

- Fields which are dereferenced in a relationship path are not written through indirect pointers (e.g. `x = &y->f; *x = z;`). The escape analysis supplies this information: for each `x = &y->f`, follow `*x` forward and look for any writes to it.

With these two filters in place, we now only need to look at direct writes to structure fields, avoiding the mess that can arise with updates `*p = ...`; if we can't determine where `p` came from, it could otherwise affect every known relationship.

Now, each write to a field can only affect the relationships which

directly mention the dereference of that field. To get the new correlations and fill in `comp_assign_value`, we need a trace `STR` for the base of the relationship (the value of type `C` for which `RFT` and `RT` are relative), and then evaluate `RFT` and `RT` after the heap write to get `NFT` and `NT`. If `NFT` or `NT` is provably `NULL`, we ignore the write; we don't care about correlations where one or the other value is `NULL` (these will not end up being useful for the casting analysis).

Usually `STR` is simple to determine. In the `timer_list` example, we only care about writes to some `x.function` or `x.data` field, and `STR` will be the associated `x`. With more complex paths in the relationship there are additional wrinkles to deal with.

Recall the file reading example from Section 6.2. Here we are interested in the relationship, for all values `f` of type `file`, between `f->f_op->read` and `f->private_data`. Again, for direct writes to some `f->f_op` or `f->private_data`, we know the exact file `*f` that is being updated. However, for writes to some `op->read` where `op` has type `file_operations*`, we do **not** know which file is being updated. Even if the write is instead to some `f->f_op->read`, there could be another file in the heap whose `f_op` field is aliased with `*f->f_op`.

Taking these concerns into account, the procedure for dealing with writes affecting `struct` type relationships is as follows:

1. Take some trace `XT` which is written, which might alias some relative trace `XRT` (both have the same trailing `fld{_,F,C}`), where `drf{XRT}` is a subtrace of either `RFT` or `RT` in a relationship on a `struct` type.

2. If there is a trace `STR` where `trace_sub(XT,STR,XRT)` **and** `XRT` contains no dereference, use that `STR` with `RFT` and `RT` to obtain `NFT` and `NT`. Ignore cases where `NFT` or `NT` are `NULL`.

3. Otherwise, if it is possible that there exists a trace `STR` where `trace_compose(STR,XRT,RT)`, then fail the relationship with the `wpoly_fail` predicate. There cannot exist such `STR` if either the write occurs in a static initializer (run before the program starts, by far the most common case for types like `file_operations`), `XT` is in a just-allocated heap location which no such `STR` has been updated to point to yet, or if `XT` and `XRT` cannot alias (via `trace_alias`) due to, e.g. incompatible fields (`x->g.f` vs. `z->h.f`).

Synchronized Writes

We can get some extra precision characterizing writes in cases where the function pointer and data traces are written in synchronization with one another, along the same code paths. For example, in the `vbi_init` example function, the following two assignments appear next to one another:

```
vv->vbi_q.timeout.function = saa7146_buffer_timeout;
vv->vbi_q.timeout.data     = (unsigned long)(&vv->vbi_q);
```

The approach described thus far adds correlations after the writes to both the `function` and `data` fields, so that the `comp_assign_value` predicates pair function `saa7146_buffer_timeout` with both the initial and new values of the `data` field, `*vv->vbi_q.timeout.data` and `vv->vbi_q`. The first of these is overapproximate since the `data` field is overwritten before the program can invoke an indirect call through the `function` field.

So, we refine our approach as follows. A write to one half of a relationship can be ignored if:

- The write will definitely be followed by a write to the other half later in the same function; the guard associated with the earlier write implies the disjunction over the guards on all the future writes (there may be multiple such writes along different paths).

- There are no calls along the paths from the earlier to the future writes where either the call can invoke the relationship's function pointer, or the call has a side effect which copies the function pointer or data to a new location. Such accesses would use the locations in the relationship while they are still in the process of being updated.

6.5.2 Writes Introducing Correlations

We need to convert each generated `comp_assign_value(...,NFT,NT)` and `func_assign_value(...,NFT,NT)` into `wpoly_receive` predicates to store in the `spoly_receive_comp` and `spoly_receive_func` sessions, respectively. The generated `wpoly_receive` need to collectively describe all the possible values of `NFT` and `NT`, and in addition need to specify concrete function names `FNPTR` rather than symbolic traces `NFT`.

There are several ways we can do this.

- Simply use the escape analysis to find all the `FNPTR` values which `NFT` could refer to (Section 6.5.2).

- Follow any transitive structural relationship between `NFT` and `NT` (Section 6.5.2). If `NFT` and `NT` were both 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 `NFT` and `NT`.

- If the current function is always invoked through an indirect call, look for structural relationships between `NFT` and the trace used to invoke that indirect call (Section 6.5.2). This approach lets us handle the `file` example, where there are relationships on files `f` not just between `f->f_op->read` and `f->private_data`, but also between `f->f_op->open` (through which `fops_open` is invoked) and `f->f_op->read`.

Each of these approaches may either fail or generate one or more sets overapproximating the values for `NFT` and `NT`. If all of the approaches fail, we must fail on the whole structural relationship; we could not capture the effect of a write updating it. Otherwise, we take the intersection of all the result sets to get the best overapproximation we can to the `wpoly_receive` values for `NFT` and `NT`.

Simple Correlations

We can usually use the escape analysis to get a set of possible functions for `NFT`: just follow `NFT` backwards and slice out the set of global functions from the result. In the simplest cases such as in `vbi_init`, `NFT` is already a named function and we get an exact singleton set. There are problems with more complex `NFT`, however.

Consider a function similar to `vbi_init` but without the write to `vv->vbi_q.timeout.function` with `saa7146_buffer_timeout`.

```
vv->vbi_q.timeout.data = (unsigned long)(&vv->vbi_q);
```

In this case we will get for `NFT` the initial value of the function pointer, `*(vv->vbi_q.timeout.function)`. There are several ways we could generalize this trace for the escape analysis while propagating backward, simplifying the trace to either values of `.function` over type

`timer_list`, values of `.timeout.function` over type `saa7146_dmaqueue`, values of `.vbi_q.timeout.function` over type `saa7146_vv`, or keeping the original trace `vv->vbi_q.timeout.function`.

Escaping using `.function` will find every function that could be assigned to **any** `timer_list`, a uselessly imprecise overapproximation. Escaping using `vv->vbi_q.timeout.function` will follow `vv` back and forth everywhere it is passed in the code, likely leading to propagation failure (hitting the propagation threshold). Escaping using the trace locations for `.timeout.function` or `.vbi_q.timeout.function` will yield the correct result, finding the only value that is assigned directly to this field chain is `saa7146_buffer_timeout`, and that there can be no indirect assigns due to taking the address of `.timeout`.

Across the many `NFT`, the best generalization method for escape propagation varies widely, so to maximize the chance that we will use the right method, we try several in parallel, yielding sets for each method that does not hit the propagation threshold, and will end up using the intersection of those that succeed.

In practice, what has worked for methods is to limit the number of fields in the trace to either one, two, or four before simplifying it to a type invariant (the escape precision levels `ep_field{1}`, `ep_field{2}`, and `ep_field{4}`). This approach is heuristic, but has the advantage that adding more methods will only improve the precision of the heuristic, at some runtime cost.

Transitive Correlations

Sometimes structural relationships are dependent on one another. Consider the following example, 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` specifies a function pointer `handler` which should be called when a specific interrupt is received. The `handler` field should be passed, among other things, the `void*` field `dev_id`, so that `.handler` and `.dev_id` share an important structural relationship. Each `irqaction` is created within `request_irq`, where the `handler` and `dev_id` are passed in by the caller. There is thus an additional important relationship between the `handler` and `dev_id` arguments to `request_irq`, and any correlations within this latter relationship should be added to the relationship on `irqaction`.

Detecting these dependencies is similar to what we did for the original indirect call sites in Section 6.4. If `NFT` and `NT` are both reachable from the same base structure, they are based on a relationship for that structure's type. If `NFT` and `NT` are different arguments to

the current function, they are based on a relationship for the current function. Any `wpoly_receive` correlations from the corresponding `spoly_comp_receive` or `spoly_func_receive` should be added for `NFT` and `NT` (unless there is also a `wpoly_fail`).

Note that this introduces read dependencies on `spoly_comp_receive` and `spoly_func_receive`. As these sessions get updated with new correlations and failures, this function may be reanalyzed, leading to a monotonic increase in the number of correlations and failures found for the relationships in the analyzed program.

Dominating Indirect Calls

Going back to the `file` example, the write we are most concerned with is that in `fops_open`, where the argument data `file->private_data` is written. We are interested in the possible values for `file->f_op->read` here, and while that value is never written in this function, we can still get information about it from the call stack. `fops_open` is never called directly, but rather only through `__dentry_open` and a few similar functions. In each of these functions we can prove that `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 that within `fops_open`, `file->f_op->open` equals `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 possible value for the `read` field is `fops_read`, which is thus the only possible value for `NFT` at the write to `private_data` in `fops_open`.

This dominating-caller technique is geared towards relationships involving function pointer tables, where there is an `open`-type method which fills in some private data for the other methods in the table to access. The technique in whole is as follows:

1. Find a function pointer trace `XFT` such that this function is only called when `XFT` is equal to a particular function `XFNPTR`. This dominance relation holds for a function `FN` when either:

   - `FN` is only called indirectly and `XFT` refers to the invoked function pointer at each parent call site. `XFNPTR` in this case is `FN`.
   - Each parent function which can invoke `FN` is itself dominated by calls where function `XFNPTR` is equal to some `YFT` within that parent (`XFNPTR` does not vary between parent functions; `YFT` may differ). There is a trace `XFT` such that within each parent, evaluating `XFT` at the call site to `FN` with `inst_trace` yields `YFT`.

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

2. Look for a structural relationship on a `struct` type between `XFT` and `NFT`. Normally the type will be a function pointer table like `file_operations`.

3. If there is such a relationship, then for each correlation within that relationship between `XFNPTR` and some `YFT`, the possible values for

`NFT` are the union over the possible values of each `YFT`. Normally each `YFT` will be a particular function; if not, resolve with the escape analysis as in Section 6.5.2.

Chapter 7

Cast Safety

We've laid the necessary groundwork to confront our main goal: can we (almost) prove the safety of the tens of thousands of downcasts in the Linux kernel? As described in Chapter 1, we want to make sure no heap location will ever be cast into and used as two different, incompatible types, thus getting partial coverage of the more general type safety property. Specifically, we want to rule out the following two situations:

- Casting the address of a variable `&x` or field `&y->f` to a new type the variable/field is incompatible with.
- Casting the same `void*` pointer (usually the result of a `kmalloc`) into multiple incompatible types.

If we try to directly address the second situation, we will end up doing a pairwise comparison of all casts which could refer to the same location, to ensure the same location is not cast to two different types.

We need to avoid this pairwise comparison, and do so by splitting the cast safety problem into two parts. First, at various points in the program we *fix* types to particular traces. The type-fixing algorithm

needs to obey the single constraint that for any execution of the program, there is no concrete location which is fixed with two different types (e.g. through two different traces which alias in that execution).

Second, for each downcast from a pointer `v` to type `str`, we need to trace backwards from the point of the cast and ensure that the target of `v` must have previously been fixed with type `str`. If we show this for all downcasts, then there can be no conflicting casts of the same location into two types in any execution of the program; since the location can be fixed at most one type in the execution, and each cast of that location can be traced back to a fixing of the location, each such cast must therefore be casting into the same type.

The type-fixing algorithm is pretty straightforward. While it could behave arbitrarily (as long as it obeys the stated constraint), any gaps where it does not find the type with which a location will be used will lead to false positives down the line. The algorithm we use follows the program's declared types, and is as follows:

- Stack- and statically-allocated variables are fixed the type with which they were declared, i.e. a declaration `str x;` fixes type `str` to the stack location of `x`.

- Heap-allocated objects are fixed the type to which they are initially cast, if there is such a cast. Memory returned by `kmalloc` and similar allocators is untyped, and typically is immediately cast in the manner of `str *x = (str*)kmalloc(sizeof str)`, in which case we will fix type `str` to the result of the `kmalloc`, at the program point where `kmalloc` returns. This process requires analysis to make sure we don't fix multiple types to the same location, and is covered in Section 7.2.

- Fields `f` of a location fixed to a structure type `str` are themselves

fixed to the type of `str.f`.

Then, the facts we are interested in proving are all of a specific form. For some trace `T`, structure type `C`, and condition `G`, we want to prove that for all the locations `L` that `T` can refer to if `G` holds, `L` was previously fixed with type `C`. We encode this as a `t_property` value.

```
type t_property ::= l_value{T:t_trace,C:string,G:g_guard}.
```

We refer to the trace `T` in a property as the *primary* trace of the property — it is the value whose type we are interested in. There may be many other traces referred to by the property, either the subtraces of `T` (`T` may contain chains of dereferences and field accesses) or values referred to by `G`, but for these other traces we do not care about their type.

The `t_property` type does not specify where `T` came from or the points at which we might consider the possible values `L` of `T`; `t_property` facts can be proved either inside a function, relative to a function call, or as a global or type invariant. These `t_property` facts are proved in a demand-driven fashion, which is described in Section 7.1. In Sections 7.2 and 7.3 we consider the seed information and individual rules which we need can prove `t_property` facts, and in Section 7.4 we discuss how to consider casts of data that originated outside the operating system (such as network packet headers and filesystem on-disk metadata).

7.1 Demand-Driven Analysis

Recall the function pointer table example we used for the polymorphic data analysis back in Section 6.2. The `read` operation of the `saa7146` filesystem driver is the `fops_read` function below.

```
// 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) {
      ...
   }
}
```

`file->private_data` has static type `void*`, so we are interested now in how to prove the correctness of this cast, that at entry to `fops_read`, the type of `file->private_data`'s target was previously fixed to `saa7146_fh`. Looking at this function, there are a variety of ways we could prove this:

1. There could be an invariant on type `file`, such that every value assigned to `file->private_data` is of type `saa7146_fh`.

2. For every call site to `fops_read`, the caller ensured a value of type `saa7146_fh` was passed in through `file->private_data`.

3. The polymorphic data analysis could have computed the correlations for a structural relationship between `file->private_data` and the function pointer used to invoke this function (which is `file->f_op->read` at every call site), finding the the values with which `fops_read` is correlated.

As it turns out, not surprisingly, the third choice is the one which will let us prove the correctness of this cast. The polymorphic data analysis finds that when `file->f_op->read` equals `fops_read`, then `file->private_data` can only be a value assigned during the driver's

open function `fops_open` (again, see Section 6.2), which is of type `saa7146_fh`.

However, it is not obvious just looking at `fops_read` which way to prove the cast's correctness will ultimately work out. There are plenty of cases where a type invariant is the most suitable, or where the extra precision of following the data back through all immediate call sites (and future parent call sites) will work out. The different methods of proving the cast's correctness mirror the different ways in which we could reason about the code, and there is no single way which will always be appropriate.

We'll address this problem by trying all the reasonable options for proving the correctness of the cast. If any of the proofs work out then the cast will be proved. Moreover, each of these proof attempts will involve numerous subproofs; proving a type invariant requires we prove facts about each write to the involved fields, proving a fact at a call site requires proving things about the callers which could reach that call site, and proving a fact about correlated data requires proofs about the function containing that correlated data. Each of these subproofs could be performed in numerous ways, with each of those ways requiring yet more subproofs.

This will continue until we find subproofs which can be proved without reliance on additional subproofs. For example, we can prove some `T` has a type `C` within a function if the type of `T` was fixed to `C` within that same function.

The individual *items* we are trying to prove are pairs between a location `h_kind_loc L` — a function, global or structure type — and a kind of fact `h_kind K` that can hold with respect to that location.

```
type h_kind_loc ::=
    hl_func{FN:string}
```

```
  | hl_glob{GLOB:string}
  | hl_comp{COMP:string}.
type h_kind.
```

Type `h_kind` will be specified later; it includes `t_property` values that must hold at, e.g. a particular point in a function, or after all writes to a field or global. Now, for any pair of a location and kind `(L,K)`, that item might be proved in zero or more ways, each way requiring zero or more subproofs of other items. These relations constitute Horn clauses, simple Datalog-style rules where there is a clause `(L,K) :- (`L_0`,`K_0`), (`L_1`,`K_1`).` when `(L,K)` is proved if both `(`L_0`,`K_0`)` and `(`L_1`,`K_1`)` can be proved. Axiom clauses `(L,K).` indicate cases where `(L,K)` is proved without requiring any subproofs.

The job of the demand driven portion of the casting analysis is to generate these Horn clauses for concrete items within the program through function, global, and type-local analysis, and to perform interprocedural analysis by making as many derivations as possible from the generated clauses.

There is, however, an infinite space of the possible items that could be proved, and we are only interested in a finite subset, those which will help us in proving the safety of the program's casts. We will fill in the space of interesting items, and the Horn clauses by which they can be proved, as a demand-driven graph search. This search is encoded within the following four sessions:

```
session scasting_try(L:h_kind_loc) containing [scheck].
session scasting_tried(L:h_kind_loc) containing [scheck].
session scasting_proved(L:h_kind_loc) containing [scheck].
predicate scheck(K:h_kind).

session scasting_derive(L:h_kind_loc)
    containing [sprint,sfirststep,snextstep].
```

```
type h_method.
predicate sfirststep(K:h_kind,
                     XL:h_kind_loc,XK:h_kind,XM:h_method).
predicate snextstep(K:h_kind,
                    XL:h_kind_loc,XK:h_kind,XM:h_method,
                    NL:h_kind_loc,NK:h_kind).
```

The first three sessions remember which items we are either going to try to prove, have already tried to prove, or have managed to prove.

- `scasting_try(L)->scheck(K)`: The clauses by which the item `(L,K)` can be proved should be generated. Either `(L,K)` is a seed item which can prove the correctness of a particular cast in the source program, or it is mentioned in the body of a clause which can be transitively used to prove one of those seed items.

- `scasting_tried(L)->scheck(K)`: We have generated the clauses by which `(L,K)` can be proved using other items. These clauses are encoded in the `scasting_proved` and `scasting_derive` sessions.

- `scasting_proved(L)->scheck(K)`: We have proved that the item `(L,K)` holds. There is a clause deriving `(L,K)` where all the items in the body of the clause have been proved.

The `scasting_derive` session encodes the Horn clauses generated for each item in `scasting_tried`. This is done by chaining together the items in the body of the clause to each other and to the item at the head of the clause. First, we introduce a new type `h_method`, which is an identifier to distinguish the different clauses used to prove each item. For a clause `(L,K) :-` $\texttt{(L}_0\texttt{,K}_0\texttt{)}$`,` $\texttt{(L}_1\texttt{,K}_1\texttt{)}$`, ...` $\texttt{(L}_N\texttt{,K}_N\texttt{)}$ associated with `h_method M`, for $i \in [0, N\rangle$ add to the `scasting_derive(`$\texttt{L}_i$`)`

session the predicate `snextstep(`K_i`,L,K,M,`L_{i+1}`,`K_{i+1}`)`. This associates the `(`L_i`,`K_i`)` with both the head and the next step in the clause; by following the `snextstep` edges, the entire remainder of the clause's body can be recovered from each item.

Then, `sfirststep` indicates the first item in the clause's body which has not yet been proved. Initially, the `scasting_derive(`L_0`)` session is seeded with `sfirststep(`K_0`,L,K,M)`, and as the first and future clauses in the body are proved, `sfirststep` is added for items further in the body until the end is reached, at which point the head of the clause is proved and added to `scasting_proved`.

This design ensures that, whenever a new item is proved, we will immediately be able to fill in `scasting_proved` for any additional items which used that new item in a subproof. Our analysis strategy will be as follows:

1. Initialize `scasting_try` with all the casts in the target program (Sections 7.2).

2. For each function, global, or type `h_kind_loc` in the program, grab a set of items from that location's `scasting_try` session which are **not** in `scasting_tried`, and run local derivation rules to find the clauses which can derive those items (Section 7.3). Add these items to `scasting_tried`.

3. For axiom clauses or clauses where all items in the body have already been proved, add the head of the clause to `scasting_proved`. For any such item `(L,K)` added, check for `sfirststep` predicates on that item for another clause deriving another item `(XL,XK)`, and add `sfirststep` for the next unproved item in the clause. If there are no more unproved items in the clause, add `(XL,XK)` to `scasting_proved` and repeat.

4. For clauses containing items which have not yet been proved, encode the missing portions of the clause in `scasting_derive` using `snextstep` and `sfirststep` as described previously. Add the items which have not yet been proved to `scasting_try`.

5. Repeat steps 2-4 until `scasting_try` equals `scasting_tried`. After a while there will be diminishing returns in running the analysis further, at which point the analysis can just be killed. Killing the analysis in this way does not impact the correctness of the items in `scasting_proved`.

A weakness here as a model for proving facts about programs is inherited from the Horn clauses themselves: this approach cannot handle mutually dependent items. The clauses introduce dependency edges between their heads and the items in their bodies, and cycles along these edges indicate items which may be mutually dependent on one another. This has not been a problem using this analysis in practice.

7.2 Casting Seed Information

The casting analysis seed information is the `scasting_explicit` session, and describes all the explicit casts which occur in the analyzed program.

```
session scasting_explicit(FN:string) containing [sexplicit].
predicate sexplicit(P:pp,TP:t_property,FIXED:bool).
```

The values `FN` and `P` give the point where an explicit cast occurs, and the property `TP` gives the trace being cast, the type being cast to, and the condition at `P` under which the cast occurs. These are straightforward to generate from the program itself.

Trickier is the value FIXED, which is true if this is the initial cast of a freshly allocated heap location, and as such fixes the type of that location to the type indicated by TP. A cast fixes the type of a trace T if it meets the following criteria:

1. T was allocated within FN by a callee CFN at an earlier point. This is given by the wfunc_allocate summary information for the salias_write_func session of CFN.

2. There are no intervening casts of T between the original allocation of T and point P within FN. If CFN is a wrapper for a more primitive allocator CCFN, this includes paths in CFN between the call to CCFN CFN's exit point.

3. Similarly, there are no calls between the original allocation of T and point P within FN where T could escape into the call and be cast to some type.

Items 1 and 2 can be modelled exactly. For item 3 we approximate whether T may be cast by a call by checking whether T is either passed directly as an argument, or if T was assigned before the call to a structure field or other location reachable from the heap. Since heap-allocated locations tend to be cast immediately after they are allocated, this is sufficiently precise.

7.3 Casting Derivation Rules

The purpose of the casting derivation rules is to construct the Horn clauses by which items (L,K) may be derived. As defined above, the location L is either a function, global, or type. The kind K which may be proved at each of these locations is defined as follows:

```
type h_kind ::=
    hk_prior{TP:t_property,P:pp,GSPLIT:bool}
  | hk_prior_inst{I:c_instr,P:pp,CTP:t_property,RECURSE:bool}
  | hk_prior_exit{TP:t_property}
  | hk_invariant{TP:t_property}
  | hk_future{FT:t_trace,C:string,P:pp,FRAME:bool}.
```

The meaning of the different values of `h_kind` are as follows.

- `hl_func{FN}`: `hk_prior{TP,P,GSPLIT}`: Property `TP` holds along all paths in the program reaching point `P` within `FN`. Where `TP = l_value{T,C,G}`, this says that the type of `T` was previously fixed to `C` when `G` holds along paths to `P`. `GSPLIT` indicates to the analysis whether or not the guard for point `P` should be examined and split up when trying to prove `TP` holds (see Section 7.3.1).

- `hl_func{FN}`: `hk_prior_inst{I,P,CTP,RECURSE}`: Property `CTP` holds along all paths in the program which invoke call `I` at point `P`. `CTP` is expressed in terms of the callee of `I`, e.g. using trace `drf{root{arg{0}}}` to refer to the first argument *at the call site*, rather than the first argument of `FN`). `RECURSE` indicates that this is a recursive function call (or tail recursive loop call, if `FN` models a loop body).

- `hl_func{FN}`: `hk_prior_exit{TP}`: Property `TP` holds along all paths in the program reaching the exit point of `FN`. `TP` is expressed in terms of the exit state of `FN`, e.g. trace `drf{root{return}}` indicates the return value of `FN`.

- `hl_glob{G}` or `hl_comp{C}`: `hk_invariant{TP}`: `TP` is an invariant and can always be assumed to hold at points where the traces

used in TP are accessed. This may be an invariant over a particular global or all values of a particular type, depending on whether the associated location is hl_glob or hl_comp.

- hl_func{FN}: hk_future{FT,C,P,FRAME}: For the current and *future* locations XT which might be pointed to by FT, the property l_value{XT,C,true} holds. This is effectively saying that hk_prior holds for the property on drf{FT}, and if FT is ever written in the future, those writes preserve the type of drf{FT}. If FRAME is true, the only locations XT to consider are those which might be pointed to by FT at future points in the current execution frame for FN, as opposed to the whole program.

The session scasting_try is initialized from the seed information in scasting_explicit by adding a hk_prior kind for each generated sexplicit cast. All other derivations are performed in an attempt to prove these initial h_kind values. In the remainder of this section we describe all the rules by which we can generate Horn clauses deriving each of the different values of h_kind.

7.3.1 Rules for hk_prior{P,TP,GSPLIT}

The rules for hk_prior form the core of the casting analysis, and most of the other kinds are expressed in terms of hk_prior. Where the property TP is l_value{T,C,G}, we need to ensure that along all paths going through P, if G holds then the location referred to by T had its type previously fixed to C.

There are two strategies we can use to handle hk_prior. First, we may be able to trivially prove that the property does or doesn't hold, generating either an axiom for the item or no clauses at all for the item.

The conditions for these trivial proofs are as follows:

- If `G` is unsatisfiable on all paths through `P`, the property trivially holds.
- If `T` is `NULL` or uninitialized, the property trivially holds.
- If the type of `T` was fixed to some type `XC` within this function on all paths prior to `P`, the property holds if `C = XC` and does not hold if `C != XC`.
- If the type of `T` was otherwise specified as `XC` by the C type system, the property may be assumed to hold if `C = XC` and not hold if `C != XC`. It is possible to specify the type of a location without actually fixing it; some declaration str *x; suggests that `x` always points to some location with type fixed to `str`, but does not guarantee it. However, if `x` points to something not of type `str`, there must have been an earlier cast of that object to type `str`, and the casting analysis will reject that cast instead.
- Similarly, if on all paths prior to `P`, `T` is cast to some type `XC`, the property may be assumed to hold if `C = XC` and not hold if `C != XC`. The presence of the cast is no guarantee that the targeted location was fixed to `XC`; this is, after all, the property we are trying to prove. If the type of `T` was not fixed to `XC`, the casting analysis will, however, reject those earlier casts.

If none of the trivial conditions applies, our second strategy for handling `hk_prior` is to identify all the places where the property could have been established, by the callers, a callee, or as an invariant, and propagate the property to each of these. We will construct a separate Horn clause whose body is all the sources in each of these possible

directions, and if the entire body of any of the clauses is eventually proved, the property is proved for our `hk_prior`.

Properties passed in by callers

Properties could have originated with the caller to the current function if they do not reference any function-local data, i.e. they do not use `uc_sum` anywhere for the unconstrained result of callee side effects. If, for each call site `XI` to this function, at point `XP` within parent function `PFN`, we can prove that `hk_prior_inst{XI,XP,TP,RECURSE}` holds for `PFN` (setting `RECURSE` depending on whether `PFN` is also the current function), then the `hk_prior` is proved.

The only additional consideration here is incorporating the results of the polymorphic data analysis. For any call site `XI/PFN` where the current function `FN` may be called indirectly through function pointer trace `PFT`, the polymorphic data analysis might identify one or more structural relationships `wpoly_call_comp(PFT,C,RFT,RT,AT)` which holds at `XI/PFN` (the case with `wpoly_call_func` is analogous).

If `spoly_receive_comp(C,RFT,RT,AT)` has no `wpoly_fail` predicate (there was no failure in generating the relationship correlations), then for each `wpoly_receive(FNPTR,XFN,XT)` in that session where `FNPTR = FN`, it is possible that argument `AT` to `FN` is equal to `XT` within some call to function `XFN`, when `FN` is called by `PFN/XI` through `PFT`.

For such call sites `PFN/XI` where there is such a non-failing structural relationship **and** the primary trace in `TP` contains `AT` as a subtrace, instead of generating `hk_prior_inst` in the body of the caller, we generate one `hk_prior` for each correlated `XFN/XT`, substituting `XT` for `AT` within property `TP`.

This reasoning contains a logical gap that we need to account for.

While the polymorphic data analysis tells us the possible values for AT in terms of XFN/XT, the primary trace of TP may be different from AT itself, and instead just contain AT as a subtrace. The contents of AT change between the point where the correlation was introduced on XT in XFN and the later indirect call to FN. There are a few cases to consider:

- If the primary trace of TP is equal to AT, the locations referred to in FN and XFN are the same and we don't need to do anything.
- If the primary trace of TP is a field of AT, i.e. drf{fld{AT,AF,AC}} for some AF and AC, we need an extra hk_future clause with FRAME = false for fld{AT,AF,AC}, ensuring its value cannot change before the call to FN.
- Otherwise, we can't do anything with the polymorphic data, acting as though there was a failure wpoly_fail associated with the structural relationship.

Properties passed out by callees

Properties could have originated with a callee if they reference data that might have been written by that callee, i.e. they contain uc_sum{I,P,_} where I/P indicates the relevant call site. Usually these uc_sum values indicate analysis imprecision which will be resolved by a separate rule (Section 7.3.1).

There are, however, cases where we are interested in propagating down into a callee. When the property includes as a subtrace either a location which we know was allocated by a callee (via the wfunc_allocate summary predicate), or the return value of a callee for which we cannot inline the write, we can replace that subtrace in the property with the corresponding callee trace, and prove the hk_prior if

we can prove the `hk_prior_exit` with the new property on all possible targets of the call (which is a singleton set for direct calls).

Global and type invariant properties

If all the traces `TP` references are rooted at global variables, the `hk_prior` is proved if we can prove `hk_invariant{TP}` for the `hl_glob` on the referenced global variables. If all the traces `TP` references contain some common structure subtrace `STR` of type `C`, the `hk_prior` is proved if we can prove `hk_invariant{RTP}` for `hl_comp{C}`, where `RTP` expresses `TP` using offset traces from `STR`.

If there are multiple such `hk_invariant` properties that could be used to prove `TP`, we will generate separate Horn clauses for each of them. This is especially common if `TP` references only a single trace (e.g. if `G = true`), in which case we will add type invariant clauses for every field access within that trace.

Guard splitting

While the `t_property` type contains a guard component to constrain the condition where the trace has the specified type, it is generally counter-productive to constrain this condition maximally. If a trace is cast to a type under the guard `A & B & C & D`, we could set the property condition to `A & B & C & D` and continue from there. The main problem this approach runs into is that it allows little reuse of the resulting derivations; if the trace is also cast to the same type under the condition `A & B & C` (i.e. omitting the `D` component) we have to generate additional derivations for this new guard, and so on for all other casts of `T` and all guards associated with points to which the property is propagated via derivations.

The alternative is to *split* the guard under which an cast occurs, break it into a set of atomic conditions (ones which do not contain `&` or `|`) such that each atomic condition is weaker than the initial condition — if the atomic condition is proved, the initial condition is proved. In the context of the earlier guard, an cast associated with `A & B & C & D` is proved if the cast can be proved safe for either `A`, `B`, `C`, or `D`. Moreover, when the second cast under condition `A & B & C` is encountered, all the earlier derivations can be reused.

Using guard splitting dramatically reduces the size and number of guards in the derivation properties, with no observed loss in precision — while a cast that requires some `A & B` to hold cannot be proved safe using splitting, we have not encountered a pattern like this in practice.

Properties clobbered by callees

A small number of fields in Linux are important to many casts, but are treated as impure and have their value clobbered (Section 5.3.2) at most call sites. If we see a cast of this field after such a call, we could not normally propagate information about that cast across the call, as we could not prove the field's value after the call equals its value before the call.

An example of this is, somewhat predictably, the `file` structure; the `private_data` field of a file is treated as impure, causing problems in functions such as the following:

```
// fs/cifs/file.c
ssize_t cifs_user_read(struct file *file, char __user *read_data,
                       size_t read_size, loff_t *poffset)
{
   ...

   xid = GetXid();
```

```
   cifs_sb = CIFS_SB(file->f_dentry->d_sb);
   pTcon = cifs_sb->tcon;

   if (file->private_data == NULL) {
      FreeXid(xid);
      return -EBADF;
   }
   open_file = (struct cifsFileInfo *)file->private_data;
   ...
}
```

The function `GetXid` (actually a macro which calls `_GetXid`) is in the giant SCC for the kernel (Section 5.3.2), and calls to it are treated as clobbering `file->private_data`, so that we cannot directly propagate the cast to `cifsFileInfo` to the entry point of `cifs_user_read` and from there to the points where the private data might have been assigned.

When a value is clobbered at a call site, and that value points **directly** to the primary trace of `TP`, i.e. the value which will be cast later, we can handle the clobber by adding an `hk_future` clause with `FRAME = true` for the field which is overwritten, ensuring that even if it does change, its value at exit from the call (and any other clobbering calls in the current function) will still be a value of the appropriate type.

7.3.2 Rules for `hk_prior_inst{I,P,CTP,RECURSE}`

The rules for `hk_prior_inst` are largely expressed with the rules for `hk_prior`. For cases where the call is not recursive (`RECURSE` is false), we use `inst_trace` to convert the callee property into all the caller properties it could correspond to. There may be zero such properties if, for example, the callee property describes the type of an argument

which is NULL at the call site; in these cases the hk_prior_inst holds as an axiom. If there are one or more caller properties TP, then the hk_prior_inst is proved if all of the hk_prior{P,TP,true} can be proved.

For recursive calls, the hk_prior_inst is proved if and only if there is a single caller property TP identical to CTP — the trace and boolean condition referred to in TP do not change over execution of the function. This handling of recursion is particularly important for the functions we use to model loop bodies; with this rule we are making an inductive argument that if a property holds at entry to the loop, and does not change over loop iterations, it holds over all iterations of the loop.

7.3.3 Rules for hk_prior_exit{TP}

The rules for hk_prior_exit are expressed entirely using the rules for hk_prior. Property TP is expressed using the exit state for the current function, so we use convert_trace to restate the property as zero or more properties expressed in terms of the function's entry state. If there are zero such properties, the hk_prior_exit holds as an axiom. If there are one or more properties XTP, then the hk_prior_exit is proved if, for the function's exit point P, all of the hk_prior{P,XTP,true} can be proved.

7.3.4 Rules for hk_invariant{TP}

An invariant on a global or type is preserved if all of the writes which can affect the traces described in the invariant preserve the invariant. For any such trace T (typically just the trace which must have a particular type, and its subtraces):

- For any point `P` in a function where the trace is directly written, the invariant is preserved if we can prove `hk_prior{P,XTP,true}`, where `XTP` is the result of replacing `drf{T}` in `TP` with the value being written to the trace.

- If the trace has its address taken, the invariant is preserved if that address will not be used for any future writes (we can determine this with the escape analysis). We are only handling invariants where the traces in the invariant are always written directly.

- Similarly, if the trace is copied somewhere, and it is **not** the primary trace of the property, the invariant is preserved if the address will no be used for any future writes. For example, if the invariant is for the type of a global trace `glob->a->b`, we need to check at any copy of `glob` or `glob->a`, but not `glob->a->b`. The former two may lead to `glob->a->b` being indirectly changed by a later write, but not the latter.

7.3.5 Rules for `hk_future{FT,C,P,FRAME}`

Proving `hk_future{FT,C,P}` requires not only that `drf{FT}` have type `C` at `P` but that future values which `FT` is overwritten with also have type `C`. Rather than trying to prove this directly with Horn clauses (which is pretty difficult), we will make assumptions about the program which we can't immediately prove, and encode those assumptions as further requirements for the program to be type safe (in addition to the initial program casts).

These assumptions are *analysis-dependent* casts: casts which are not explicit in the program but which are added to `scasting_explicit` and treated exactly like any other cast as the casting analysis proceeds.

Now, for any `hk_future` we need to establish three constraints:

- `hk_prior` holds for the possible values of `FT` at `P` (the trace currently has type `C`).

- If `FT` is a field trace `fld{_,XF,XC}`, then updates to that field are *type preserving*: whenever the field is written, either the old value is `NULL` or uninitialized, or the type of the new value is the same as the type of the old value. We establish the field is type preserving by adding analysis dependent casts immediately before each write of the field, to check the compatibility of the old type with the new type.

  If `FT` is **not** a field trace, we cannot prove the `hk_future` clause at all (in previous sections, `hk_future` clauses were only generated for field traces).

- If the value of `FT` at point `P` might be `NULL` (or uninitialized), we need to assert the type of `drf{FT}` as `C`, to avoid the problem of considering the `NULL` value as having multiple types. Consider the following toy program:

```
void foo(base_str *s)
{
  s->data = NULL;
  bar(s);
  str1 *data1 = (str1*) s->data;
}

void bar(base_str *s)
{
  s->data = (str2*) malloc(sizeof(str2));
}
```

If we prove the cast to `str1` with an `hk_future` clause at the call site to `bar`, we will add as an assumption an analysis dependent cast at the assignment within `bar`, which can also be proved since the rules for `hk_prior` allow `NULL` to have any type.

Adding an assertion at the call site to `bar` that `s->data` has type `str1` (this is done with a new predicate in `scasting_explicit`) causes the casting analysis to act as though there is an assignment at that call site to `s->data` of a value of type `str1`, which keeps us from proving the assignment in `bar` is type preserving.

If `FRAME` is true, we only need to worry about type preserving assignments within the callees of the current function and can thus add the assertion at point `P`. Otherwise, we need to add the assertion at the point where `FT` was originally allocated, failing to prove the `hk_future` if we cannot find the allocation.

While in this chapter we do not distinguish the original and analysis-dependent casts, when giving the analysis results in Section 9.6 we only describe the original program casts. There are a total of 446 analysis-dependent casts added to the code (an increase of 1.5%), of which we are able to prove 327 (73%, about the same as the proof rate for the program casts).

7.4 Internal Casts vs. External Casts

The Linux kernel along with all other operating systems has to interact with a wide variety of network interfaces, hard disks, and other storage and I/O devices. The data read in from these devices is often structured, including such things as packet headers and filesystem organizational structures. These are declared using `struct` in various

kernel headers, and when data is read in from an external device that data is cast to the `struct` before being used.

We'll call these sorts of casts *external* casts. An external cast is one where the location that is the target of the cast originated from a device or from userspace, and the writes to the location that filled in the structure were not performed directly by the currently running operating system. This contrasts with *internal* casts, where the location being cast was created and written to by the operating system, and has lived either in memory or in a paging file on disk ever since.

For the purposes of type safety and for this analysis we are only interested in internal casts. Even in cases where an external cast is performed, data that originated outside the filesystem cannot be assumed to follow any particular data layout, and will be (or ought to be) heavily sanity-checked to ensure all the data makes sense within the layout of the type to which the external cast is made. Without this sanity checking, a malicious agent or malfunctioning device could take over or bring down the whole system by passing in some poorly formed data.

In contrast, the data read after making an internal cast can only be that which was written by some other function within the running kernel. In general, no sanity checking needs to be done here and the kernel will assume after the cast that the data stored in the fields of the structure is well formed. This assumption is correct provided the kernel is type safe.

In Section 7.4.1 we give an example of an external cast illustrating the difficulties in identifying them, and in Section 7.4.2 we explain why we don't really have a way to identify the external casts in Linux.

7.4.1 External cast example

An important use of external casts is during interaction between a filesystem driver and the underlying disk. Besides the actual contents of the user's files, almost all the data stored on disk is heavily structured metadata describing file properties, directory hierarchies, and so forth. The filesystem driver must maintain and keep in sync both the on-disk structures and their in-memory counterparts. This can be seen in the `ext2_fill_super` function below, which is called whenever an ext2 filesystem is mounted.

```
// fs/ext2/super.c
static int ext2_fill_super(struct super_block *sb,
                           void *data, int silent)
{
   struct buffer_head * bh;
   struct ext2_sb_info * sbi;
   struct ext2_super_block * es;
   ...
   unsigned long logic_sb_block;
   unsigned long offset = 0;
   ...

   sbi = kmalloc(sizeof(*sbi), GFP_KERNEL);
   if (!sbi)
      return -ENOMEM;
   sb->s_fs_info = sbi;
   memset(sbi, 0, sizeof(*sbi));

   ...

   if (!(bh = sb_bread(sb, logic_sb_block))) {
      printk ("EXT2-fs: unable to read superblock\n");
      goto failed_sbi;
   }

   es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
```

```
   sbi->s_es = es;
   sb->s_magic = le16_to_cpu(es->s_magic);

   if (sb->s_magic != EXT2_SUPER_MAGIC)
      goto cantfind_ext2;

   ...
}
```

The purpose of `ext2_fill_super` is to create and fill in an in-memory representation `sbi` of the filesystem's superblock (a structure storing filesystem-wide settings and information), so that later on the driver can query `sbi` directly instead of reading from disk (we will cover superblocks in greater detail in Section 9.6.3). This is done by performing the following steps in the above function:

1. Allocate and zero out `sbi`.

2. Get a `buffer_head bh` for the portion of the disk containing the on-disk superblock. `sb_bread` uses the virtual memory subsystem to directly map a range of virtual addresses to the disk, storing this range at `bh->b_data`. When `bh->b_data` is accessed, the disk blocks containing the superblock will be read and the resulting data paged in.

3. Cast the pointer `bh->b_data + offset`, which again is referring to data paged in from disk, into type `ext2_super_block`, which describes the layout of a well-formed on-disk ext2 superblock.

4. Sanity check the `es->s_magic` value (which is `EXT2_SUPER_MAGIC` if the filesystem is indeed ext2).

5. Sanity check and store in `sbi` the numerous other fields in `es` (omitted).

From an analysis perspective, the key problem here is that the cast of `bh->b_data + offset` looks exactly like any other cast in the kernel. Since data on a disk or a network interface or device is often accessed through specially mapped virtual addresses, there does not seem to be an easy way to distinguish these accesses from accesses to other kernel-space data. This is an important problem in and of itself, and we need a good solution in order to reliably differentiate internal from external casts.

7.4.2 Identifying external casts

Unfortunately, we don't have a good mechanical way to identify external casts. We can, though, use a crude filter to drop almost all the external casts, as well as a portion of the internal casts. For any cast to a structure type `str`, if `str` does not contain any pointer fields, either directly or transitively through its substructures, we will ignore the cast and will not try to verify its safety. Pointers are virtual memory addresses, and devices external to the kernel do not store them in structures or otherwise care about them (there are a couple exceptions here where pointers are used in structures to store physical addresses that need translation).

This filter throws away 16143 casts of the 44910 total downcasts of pointers to structure types in the kernel, or 36%. This includes all casts to 3122 different structure types.

From inspection, the majority of these casts do not seem to actually be external casts, and we can prove their correctness with close to the same accuracy as we can the casts to pointer types.

Chapter 8

Analysis Implementation

In this chapter we present and give performance metrics for all the analyses that are executed in order to build up the necessary information for the final casting analysis. We divide up this process into a list of passes, where each pass takes as input the databases generated by the previous passes (except the initial frontend), and produces as output one or more new databases for use by future passes. A few passes modify one or more of their input databases (these cases are mentioned in Section 8.1); however, When one pass finishes it will not need to be run again later on.

In Section 8.1 we briefly describe each of the passes performed and their output databases, up to and including the final casting analysis. In Section 8.2 we give timing data for each of the passes, and in Section 8.3 we give timeout data for each of the passes.

8.1 Analyses

Except for the initial frontend, all of the passes are a list of one or more Calypso `.clp` analysis files, which are fixpointed together: if analysis of one function/type/global depends on the analysis result of another function/type/global, it will be repeatedly reanalyzed until the databases stop changing.

The analysis passes are described in the following subsections, in the order in which they are performed.

8.1.1 CIL frontend analysis

This pass reads preprocessed C files and produces databases encoding their syntax (Section 2.2).

- `cil_body.db`: Syntax for all function definitions.
- `cil_comp.db`: Syntax for all struct/union definitions.
- `cil_enum.db`: Syntax for all enum definitions.
- `cil_glob.db`: Syntax for all global variable definitions (excluding static initializers).
- `cil_init.db`: Syntax for all global variable static initializers.
- `cil_type.db`: Syntax for all typedef definitions.

8.1.2 sumbody

This pass converts loops to tail-recursive functions and produces loop-free CFGs for each function/loop (Section 2.2.2).

- `cil_sum_body.db`: Syntax for a function definition, and loop-free CFGs for either its outer body or an inner loop. One database entry for each outer function body and inner loop.

8.1.3 funptr

This pass runs the function pointer analysis from Chapter 4 and identifies the possible targets for each indirect call in the program.

- `sum_init_assign.db`: All assignments that occur in the static initializer for each global variable, indexed by that global.
- `sum_init_assign_field.db`: All assignments to a particular field that occur in **some** static initializer, indexed by the field name.
- `sum_funptr_prop.db`: Intermediate propagation data.
- `sum_funptr.db`: Possible targets of each indirect call.

8.1.4 sumcallers

This pass generates the sets of possible callers and callees of each function.

- `cil_sum_caller.db`: For each function, the possible callers of that function.
- `cil_sum_callee.db`: For each function, the possible callees of that function (through either a direct or indirect call).

8.1.5 callgraph

This pass generates a compact representation of the transitive callees of each function within the callgraph specified by `cil_sum_callee.db`. This is used to answer 'can `foo` transitively call `bar`' queries.

- `cil_callee_context.db`: Callees from `cil_sum_callee.db` with some optional calling context information, for a partially context-sensitive callgraph. `funptr_refine` below can generate this context information; if this file is not used, the callgraph generated here is context-insensitive.
- `cil_all_callee.db`: Transitive callees of each function. This is a (generally small) list of functions which are not called by the big callgraph SCC (Section 5.3.2), a predicate indicating whether the big SCC itself is called, and a predicate indicating whether the big SCC calls this function.

8.1.6 init_aliasing

This pass indexes all assignments and other operations performed by the program so that demand-driven escape queries can be performed (Chapter 5).

- `salias_escape.db`: For each trace location, all the assignment edges in the program which have that trace as their source or target.
- `salias_used.db`: All points where each trace location is used for a read, write, index, field access, and so forth.

- `salias_init.db`: All assignments occurring in static initializers. This is similar to but more precise than the `sum_init_assign.db` computed by the function pointer analysis.
- `salias_rfld.db`: All points where each field has its reverse field `rfld` taken (getting the base structure from the field pointer).
- `salias_fld.db`: All points where each field is accessed.

8.1.7 init_readonly

This pass generates escape analysis caching/optimization data storing any per-function read-only data found (Section 5.2.3).

- `salias_func_readonly.db`: For each function, which traces passed into that function have been determined to be read-only.

8.1.8 init_relative

This pass generates escape analysis caching/optimization data storing where each field is written in a relative or non-relative fashion (Section 5.2.3).

- `salias_comp_relative.db`: For each field `g`, stores whether there are non-relative writes to the field (e.g. `y->g = ...`), as well as all relative writes to the field (e.g. `z->f.g = ...`).

8.1.9 usemod_comp

This pass identifies and marks all impure and initialization writes to each field (Section 5.3.2).

- `salias_write_comp.db`: Write information for each field.

8.1.10 usemod_func

This pass identifies and marks all impure functions, and the possibly written traces for semi-pure functions (Section 5.3.2).

- `salias_write_func.db`: Write information for each function.

8.1.11 funptr_refine

This pass overwrites `sum_funptr.db` with a more precise set of targets based on the local memory and escape analyses (which now have all their input databases filled in). For most indirect calls this doesn't make a difference, but the extra precision (especially the extra precision of the escape analysis over the function pointer analysis) is helpful in some cases.

8.1.12 memory_remote

This pass runs the local memory analysis over each function and loop, generating and storing all information needed to do local memory analysis queries on that function and loop. This allows both caching of the results to (slightly) speed up later analysis, and more importantly allows memory analysis queries on a function to be performed even if that function is not being currently analyzed (e.g. when analyzing `foo`, memory queries on the values of traces in `bar` can be performed).

- `smemory_remote.db`: For each function and loop, stores all data needed to do queries on the local memory analysis (e.g. `val`, `inst_trace`, etc.) on that function/loop. This includes the `guard`, `eval`, `lval`, `assign`, and a few other predicates. `val` itself is not stored, as this predicate is generated on demand.

8.1.13 init_casting #1

This pass runs an initial pass of the casting seed analysis to generate seed information for use by the polydata analysis (Section 6.4).

- `spoly_call_try.db`: For each function, traces within that function which will be cast to some type, and in which we want the polydata analysis to find structural relationships describing.

8.1.14 init_poly_data

This pass generates all seed information used by the polymorphic data analysis (Section 6.4).

- `spoly_call.db`: Structural relationships holding at each indirect call.
- `spoly_comp.db`: Structural relationships we are interested in for each type.
- `spoly_func.db`: Structural relationships we are interested in for each type.

8.1.15 poly_data

This pass runs the main polymorphic data analysis (Section 6.5), adding additional entries to `spoly_comp.db` and `spoly_func.db`, and generating the following databases:

- `spoly_receive_comp.db`: Correlations for each relationship in `spoly_comp.db`.

- `spoly_receive_func.db`: Correlations for each relationship in `spoly_func.db`.
- `spoly_receive_attribute.db`: Correlations found for `sysfs` attributes (see Section 9.5.3).

8.1.16 init_casting #2

This pass generates all seed information used by the casting analysis (Section 7.2).

- `scasting_explicit.db`: All casts in the program we will be checking, and all points where the type of a value is fixed.
- `scasting_explicit_field.db`: For fields whose value or contents are cast, the different types and locations of those casts.
- `scasting_allocate_untyped.db`: Functions returning freshly allocated data which do not cast the value before returning.
- `scasting_try.db`: `h_kind_loc`/`h_kind` facts we should try to prove (initially these are just the `scasting_explicit.db` casts).
- `scasting_post.db`: Stores data on casting results for a web-based user interface.

8.1.17 casting

This pass runs the main casting analysis (Chapter 7), adding additional entries to `scasting_try.db` and `scasting_post.db`, and generating the following databases:

- `scasting_explicit_safe.db`: Casts in `scasting_explicit.db` which were proved safe.
- `scasting_tried.db`: Each item from `scasting_try.db` which the analysis tried to prove.
- `scasting_derive.db`: Derivation steps for each item added to `scasting_tried.db` which depends on proofs of one or more other items in `scasting_try.db`.
- `scasting_proved.db`: Each item from `scasting_tried.db` which was proved.

8.2 Analysis Performance

The following table gives timing information for each of the analyses described in Section 8.1. All runs were performed on a cluster; for the main casting analysis 70 cores were used, and for all other analyses 50 cores were used (see Section 2.4.1).

The 'hours' column gives the real time the analysis takes to complete on the cluster, 'CPU h' the total time spent across all cores, and 'CPU h used' the portion of 'CPU hours' where cores were doing useful work (running the Calypso interpreter), rather than communicating with or waiting on communication from the server.

Analysis	Hours	CPU h	CPU h Used	% Used
`sumbody`	0:19	15:36	2:26	0.16
`funptr`	0:34	28:01	9:24	0.34
`sumcallers`	0:21	16:50	1:49	0.11
`callgraph`	0:34	26:59	1:02	0.04
`init_aliasing`	2:26	120:13	11:44	0.10
`init_readonly`	1:20	66:15	52:50	0.80
`init_relative`	0:28	22:55	10:07	0.44
`usemod_comp`	0:58	47:55	32:47	0.68
`usemod_func`	1:13	59:41	32:17	0.54
`funptr_refine`	1:12	59:00	40:40	0.69
`memory_remote`	0:46	37:28	16:22	0.44
`init_casting #1`	0:44	35:18	18:06	0.51
`init_poly_data`	0:51	38:48	21:40	0.56
`poly_data`	2:51	140:46	108:25	0.77
`init_casting #2`	0:46	35:17	18:19	0.52
`casting`	16:16	1087:23	304:25	0.28

The efficiency with which an analysis runs on the cluster is primarily dependent on how much work it does versus how many summaries and other sessions it updates. At one end of this scale are the `callgraph` and `init_aliasing` analyses, which do very little work but write to many sessions; the former analysis writes a session for every possible callee of the function it is analyzing, while the latter may write to several different sessions for every single assignment in the function it is analyzing. Writing to all these sessions incurs an overhead where the writes incoming from many different cores have to be merged together

to give the final contents of the session. Consequently, adding cores has little benefit and even with 50 cores in use only a small speedup over the single core performance is realized.

At the other end of this scale, however, are the analyses that do the most work and are thus the ones we most need to parallelize, such as the `poly_data` analysis. This is the second most expensive analysis (after the casting analysis itself) as it looks at all structural relationships each write might affect and has to try several different approaches to characterizing those effects. However, it writes to and reads from relatively few sessions and most of its time is spent in the Calypso interpreter rather than talking to the server.

In the middle of this scale is the `casting` analysis itself, whose running time exceeds that of all the other analyses combined. Similar to the `poly_data` analysis, many proof strategies need to be considered for each `h_kind` value the casting analysis tries to prove, and for most functions the casting analysis will end up trying to prove several hundered different `h_kind` values. However, most of these proof strategies depend on proofs in other functions, types or globals, and analyzing a single function may require writing to hundreds of sessions in the `scasting_derive.db` database, dragging down the overall efficiency of the analysis.

However, the casting analysis has a nice property that the soundness of its result does not depend on termination; the proofs it has constructed at any given point are all valid. Indeed, the space of proofs it could try to construct is of unbounded size, and if left to run it will likely never terminate. For the run presented above and prior runs of the analysis we simply ran it overnight, killed it and used the results produced up to that point.

Running the casting analysis for long times can produce additional

returns, but these diminish quickly. Of the casts which it would eventually prove safe, the casting run above constructed 50% of those proofs in the first 11% of its running time, 80% of those proofs in the first 19%, 95% in the first 29%, and 99% in the first 63%. For the 6 hours after that 63% mark, an average of 36 new proofs per hour were being constructed (see Section 9.6 for the total numbers of proofs and other casting information constructed).

8.3 Analysis Timeouts

The following table gives timeout information for each of the analyses described in Section 8.1. Since all analyses written with Saturn run at the granularity of a function, type, or global, the Calypso interpreter can time out on any given function/type/global without affecting the analysis of other functions/types/globals. However, timing out will omit from the database any session changes made while analyzing that function/type/global (or any that would have been made after the timeout), and for many analyses this will compromise the soundness of their result.

The 'sound' column gives whether the information computed by the analysis is sound even if there are timeouts, the 'TO #1' column gives the number of timeouts in the initial iteration of the analysis fixpoint (the first time each function/type/global is scanned), the 'TO #2+' column gives the number of timeouts in any subsequent iterations (if the analysis needs to fixpoint), and 'TO %' gives the timeouts as a fraction of the number of functions and loops in the code base.

Analysis	Sound	TO #1	TO #2+	Total	TO %
`sumbody`	No	0	n/a	0	
`funptr`	No	6	0	6	0.00005
`sumcallers`	No	0	n/a	0	
`callgraph`	No	0	n/a	0	
`init_aliasing`	No	7	n/a	7	0.00006
`init_readonly`	Yes	44	n/a	44	0.00035
`init_relative`	No	5	n/a	5	0.00004
`usemod_comp`	No	56	n/a	56	0.00045
`usemod_func`	No	116	4	120	0.00095
`funptr_refine`	Yes	150	n/a	150	0.00119
`memory_remote`	No	117	n/a	117	0.00093
`init_casting #1`	Yes	134	0	134	0.00107
`init_poly_data`	Yes	195	n/a	195	0.00155
`poly_data`	No	436	86	522	0.00415
`init_casting #2`	No	132	0	132	0.00105
`casting`	Yes	276	602	878	0.00698

For the early analyses computing flow-insensitive function pointer, callgraph, or aliasing information, the timeout rate is miniscule. The first analysis which uses the path-sensitive local memory analysis (Section 3.3) is `usemod_func`, and after this point there is a set of at least 100-150 functions which time out in all of the analyses. The core of this set is fairly stable — about 0.1% of the functions in Linux confound the local memory analysis, and cause it to generate an enormous number of `val`, `eval`, etc. predicates and timeout before we even get to the rules for the client of the local memory analysis. Typically the amount of

local memory analysis information is linear in the function size, but it is worst case exponential and this bad behavior is exposed on this small set of functions.

Most of these functions are related either to cryptography or to the internal device- or disk-facing interfaces of drivers and filesystems. While not analyzing these functions does of course affect the soundness of the overall results, these functions generally aren't relevant to the casting behavior of the system.

More important are the extra timeouts that the `poly_data` analysis has over the other local-memory based analyses. These extra timeouts are due to inordinate time processing the writes in the function that can lead to marking a structural relationship as failed, or to adding correlations on a structural relationship. When the polydata times out, these writes will not be considered at all in the final summaries. However, we will see in Section 9.5 and Section 9.6 that relatively few of the relationships we consider in the `poly_data` analysis will end up being used to prove casts; we have looked through many of the roughly 200 extra functions that timed out on the `poly_data` analysis but not other analyses, and most are involved only with structural relationships that will **not** be used by the `casting` analysis.

The `casting` analysis itself has the most timeouts. Fortunately, these timeouts do not affect the soundness of the constructed proofs. If proving a particular cast requires us to generate a sub-proof for some function, and analysis of that function times out, we will not have the sub-proof and can't prove the original cast. These timeouts are a minor source of false positives in the overall casting results.

Chapter 9

Analysis Evaluation

The analysis passes described in Section 8.1 run in sequence, each pass consuming summary information generated by previous analyses, and producing new summary information to be consumed in turn by the future analyses.

Besides the final casting analysis, for which we can fix a definite number on how many casts we found a proof for, the quality of the remaining intermediate passes is largely determined by how suitable and precise their summaries are as input to the later passes. A certain minimum level of precision will be needed from each pass to ultimately prove the casts, and if that precision is not attained then the poor information will cascade through and poison the results of the remaining passes.

There is a cost, though, to making a pass **more** precise than is needed by the subsequent passes. Increasing precision will not just increase computation time but, we have found, will in general (though not absolutely) make the analysis more complex, harder to understand and reason about, and less predictable in its behavior.

We want to get the precision of an analysis close to what is needed

from it, and for cases where the code is too complex for us to infer the information we will ultimately need, we will fill in the gaps with annotations.

These annotations are trusted, i.e. they are assumed correct by the passes and if they are in fact wrong we will not detect the problem. Since they are describing cases we can't model, they must be trusted. However, there are relatively few annotations in use by the various analyses, and we can use the number of annotations the analyses in a pass require as a measurement of how good a job that analysis does of modeling the code.

In Section 9.1 we give a general overview of these annotations, in Sections 9.2 through Sections 9.5 we describe the annotations used and other results for the analyses leading up to the casting analysis, and in Section 9.6 we describe the results of the casting analysis itself.

9.1 Annotation Overview

Annotations in Saturn work somewhat differently than in most program analysis systems. Saturn does not have support for in-code annotations, where a user adds machine-readable documentation to the code in the form of type qualifiers or specially formatted comments. Instead, annotations are added in the Saturn analyses themselves, in the form of logic programming rules that are custom-tailored to the program being analyzed.

While in a sense every analysis in this project has been tailored to Linux, by its virtue of being our sole target system, most of the algorithms outlined can be used as-is on other systems written in C or similar languages. We define an annotation here as a logic programming rule that targets specific functions, types, or fields in the target system.

The decision for when to use an annotation instead of writing a general purpose analysis or ignoring a particular pattern (using a weak overapproximation) is highly subjective. The annotations we added were largely due to either patterns that occur only once or very few times, or to patterns which occur only occasionally but are complex and involve a lot of code, making them difficult to write analyses for.

In all, there are 360 such annotations used by the analyses in the passes described in Section 8.1. The following table shows the number of annotations used for each pass.

Analysis	Annotations
`sumbody`	0
`funptr`	44
`sumcallers`	0
`callgraph`	1
`init_aliasing`	33
`init_readonly`	3
`init_relative`	0
`usemod_comp`	25
`usemod_func`	16
`funptr_refine`	0
`memory_remote`	0
`init_casting #1`	0
`init_poly_data`	5
`poly_data`	177
`init_casting #2`	47
`casting`	9

The remaining sections of this chapter include descriptions of many of the annotations incorporated into these passes.

9.2 Function Pointers

The function pointer analysis from Chapter 4 is the simplest analysis we use but requires relatively few annotations; this analysis illustrates the value of finding a compromise between analysis complexity and annotation burden.

As was described in Chapter 4, we will propagate indirect call targets along assignment edges between variables (globals, locals, arguments, return values), fields, and array indexes of these. If we ever find additional indirection when propagating backward from an indirect call, or a cast of a function from some other type, the analysis will fail and require an annotation.

Using this algorithm on Linux finds 98836 possible indirect call targets and fails on 46 assignments. After annotating these failures the total number of indirect call targets found is 99118, an increase of 282 targets.

These call targets cover the possible callees of 11976 indirect call sites. Of these call sites, 2347 (20%) have only a single possible target (many of these call sites are guarded, so that the function pointer is also allowed to be `NULL`), 8039 (67%) have five or fewer possible targets, and 10480 (88%) have twenty or fewer possible targets. There are some call sites with many targets; 157 call sites have at least 100 targets, and 21 call sites have at least 500 targets.

Even so, the large number of targets identified for these sites seems to reflect the actual behavior of the code, not analysis imprecision; these sites really can call that many different functions. For example, the

function containing the call site with the most targets is `dev_attr_show`, shown below:

```
// drivers/base/core.c
static ssize_t
dev_attr_show(struct kobject * kobj,
              struct attribute * attr, char * buf)
{
   struct device_attribute * dev_attr = to_dev_attr(attr);
   struct device * dev = to_dev(kobj);
   ssize_t ret = -EIO;

   if (dev_attr->show)
      ret = dev_attr->show(dev, dev_attr, buf);
   return ret;
}
```

`dev_attr_show` is used by the Sysfs filesystem to display information about currently mounted drivers. Any driver that wants to make its information accessible to the user must provide one or more (usually many more) `show` methods, and store those in `device_attribute` structures which will eventually be accessed by `dev_attr_show` (for more information on Sysfs, see Section 9.5.3). There are a total of 1112 such `show` methods, and as such `1112` possible targets of this indirect call.

Note that if we were trying to build a context-sensitive call graph (see Section 5.3.2), we could **not** narrow down the possible targets of this indirect call by examining the calling context of `dev_attr_show`. Any of the `show` methods could be the target here regardless of where `dev_attr_show` is called.

Some of the indirect call target sets can be improved. The later `funptr_refine` pass uses the escape analysis to find a smaller set of targets for each call, if possible. This trims away 13913 call targets (14%) of the original call targets, affecting 881 (7.4%) of the indirect

call sites. Some of these sites will be important for later analyses, for example the indirect call in the `ctnetlink_del_conntrack` function from Section 1.2.

Still, by itself the function pointer analysis does a pretty good job. The main problem is the need to fix the points where it fails with annotations. While many of the analysis failures could be addressed by a smarter analysis, there are some that will remain difficult to handle fully automatically. Most of the failures fall into one of two categories, described in the following sections.

9.2.1 Casts to `void*`

If a function pointer is cast from a `void*` then a failure will result. In simple cases the `void*` is always a function, and we can perform our regular backward propagation to find the targets, such as the `func` parameter in the following example:

```
// drivers/scsi/qla2xxx/qla_os.c
static inline void
qla2x00_start_timer(scsi_qla_host_t *ha, void *func,
                    unsigned long interval)
{
   init_timer(&ha->timer);
   ha->timer.expires = jiffies + interval * HZ;
   ha->timer.data = (unsigned long)ha;
   ha->timer.function = (void (*)(unsigned long))func;
   add_timer(&ha->timer);
   ha->timer_active = 1;
}

// drivers/scsi/qla2xxx/qla_os.c
int qla2x00_probe_one(...)
{
   ...
   qla2x00_start_timer(ha, qla2x00_timer, WATCH_INTERVAL);
```

```
   ...
}
```

In more complex cases the `void*` is part of a location which could hold many non-function values. We do not want to conflate the function pointer with the non-function values that location could have.

```
// include/linux/moduleparam.h

typedef int (*param_set_fn)(const char *val,
                            struct kernel_param *kp);
typedef int (*param_get_fn)(char *buffer,
                            struct kernel_param *kp);

struct kernel_param {
   const char *name;
   unsigned int perm;
   param_set_fn set;
   param_get_fn get;
   void *arg;
};

// drivers/char/ipmi/ipmi_watchdog.c

static int set_param_str(const char *val, struct kernel_param *kp)
{
   action_fn  fn = (action_fn) kp->arg;

   ...
   rv = fn(valcp, NULL);
   ...
}

static int get_param_str(char *buffer, struct kernel_param *kp)
{
   action_fn fn = (action_fn) kp->arg;

   rv = fn(NULL, buffer);
   ...
```

```
}

module_param_call(action, set_param_str, get_param_str,
                  action_op, 0644);
module_param_call(preaction, set_param_str, get_param_str,
                  preaction_op, 0644);
module_param_call(preop, set_param_str, get_param_str,
                  preop_op, 0644);
```

In general the `arg` field of a `kernel_param` could have thousands of different values, but the only possible values for `kp->arg` when passed into `set_param_str` and `get_param_str` are the functions `action_op`, `preaction_op`, and `preop_op`. This is due to a structural relationship (Section 6.3) on `kernel_param` between its `set`/`get` and `arg` fields, and `module_param_call` is a macro which constructs a `kernel_param` structure with the specified `set`, `get`, and `arg` fields.

While the `poly_data` pass can compute this and precisely characterize the possible arguments to `set_param_str` and `get_param_str`, this pass requires the function pointer information to execute, as well as far more summary information that itself will depend on the function pointer information.

9.2.2 Non-static arrays of function pointers

Functions might not be pointed to directly by variables or fields, but through arrays as well. In cases where the arrays have their address taken and are accessed indirectly the function pointer analysis will fail. (Cases where the array is always accessed by name are handled without annotation, e.g. `fnptr arr[10]; ... arr[i](data);`).

```
// net/core/wireless.c
static inline iw_handler get_handler(struct net_device *dev,
                                     unsigned int cmd)
```

```
{
   ...

   index = cmd - SIOCIWFIRST;
   if(index < dev->wireless_handlers->num_standard)
      return dev->wireless_handlers->standard[index];

   index = cmd - SIOCIWFIRSTPRIV;
   if(index < dev->wireless_handlers->num_private)
      return dev->wireless_handlers->private[index];

   return NULL;
}

// drivers/net/wireless/hostap/hostap_ioctl.c
static const iw_handler prism2_handler[] =
{
   (iw_handler) prism2_ioctl_siwfreq,
   (iw_handler) prism2_ioctl_giwfreq,
   ...
};

// drivers/net/wireless/hostap/hostap_ioctl.c
static const iw_handler prism2_private_handler[] =
{
   (iw_handler) prism2_ioctl_priv_prism2_param,
   ...
};

// drivers/net/wireless/hostap/hostap_ioctl.c
const struct iw_handler_def hostap_iw_handler_def =
{
   ...
   .standard            = (iw_handler *) prism2_handler,
   .private             = (iw_handler *) prism2_private_handler,
   .private_args        = (struct iw_priv_args *) prism2_priv,
   .get_wireless_stats  = hostap_get_wireless_stats,
};
```

In this case the function pointer returned by `get_handler` will be

called in a subsequent function. This function pointer is always fetched from either the `standard` or `private` fields of a `iw_handler_def`, and these fields are always initialized via static initializers in the same manner as `hostap_iw_handler_def`. Handling this code automatically would require tracking not just assignments to the contents of the arrays of function pointers, but assignments to the `standard` and `private` fields themselves. The escape analysis does this, but itself requires function pointer information in order to run.

9.3 Memory Model

There are several facets of the local memory model where the assumptions we made in the design depend heavily on the code base's actual behavior.

- Virtually all pointer arithmetic follows a few well defined patterns, and can be modelled given the assumption of type safety within the memory model.
- Almost all possible aliasing between traces at entry to a function is straightforward to disprove, by looking at the types of the traces or where the traces might have propagated from.
- Impure functions and fields as used for clobbering traces at call sites are a good approximation; there really are many functions which can write to almost anything, and many fields which can be written almost anywhere.

In the following sections we discuss in turn each of these aspects of the memory model, and the additional annotations used for the places where they are deficient.

9.3.1 Pointer arithmetic

As discussed in Chapter 1, when we assume type safety we are assuming that pointer arithmetic is not used to move between the fields of a structure, such as to access those fields in turn without explicitly naming them. This restriction does not rule out the most common uses of pointer arithmetic, including:

- Iterating through an array or heap-allocated buffer of data, so long as the iteration does not overflow past the end of the array and into another structure field or other memory location.
- Using `container_of` to obtain the base address of a structure from the address of one of its fields.
- Using arithmetic to skip the entire length of a structure to access data stored after the last field. This is used by several important structures in Linux; for an example see Section 9.6.1.

There are, however, cases where the programmer intends to iterate through the fields of a structure by incrementing a pointer. The only such cases we have found are the `memcpy` and `memset` routines endemic to C and C++ programs, along with similar but more specialized functions such as `copy_from_user` and `copy_to_user` (for copying memory between the user and kernel address spaces).

Throughout the Linux kernel, `memcpy` is called 5740 times, and `memset` is called 4887 times (we group `memmove` in with `memcpy`; this accounts for 267 calls). Each of these call sites is overwriting some section of kernel memory, including potentially the fields of a structure. How well does our type safety definition handle the actual uses of `memcpy` and `memset`?

There are four categories of calls to `memcpy` and `memset` which we consider. The only one which we model within the memory analysis is the first. For each category we give the number of calls which fit in the category.

1. Calls which are definitely updating a structure of a particular type, copying data to all fields or setting all fields to zero. We can identify these as the length of the copy will be `sizeof(str)` for some struct `str`, and the updated location will be of type `str`. Within the local memory analysis we add annotations for the `memcpy` and `memset` functions to model these calls precisely, as if they were assigning to each field of the target in turn.

   `memcpy`: 628 calls, `memset`: 2197 calls. The discrepancy between the number of calls to `memcpy` and `memset` within this category is due to the common practice of zero'ing out a structure with `memset` immediately after allocating it.

2. Calls which are updating a buffer which we can prove will never be used as a structure. These could be strings, arrays of integers, or other unformatted data. We can find the possible uses of an updated buffer with the escape analysis, following the buffer forward and looking for uses of any fields.

   `memcpy`: 3379 calls, `memset`: 1603 calls.

3. Calls which are updating a buffer that can only be used as a structure that does not contain pointer fields. This is the same rough filter we used in the casting analysis (Section 7.4) to filter out uses of structured data coming from network interfaces, disk drives, and other attached devices.

   `memcpy`: 970 calls, `memset`: 545 calls.

4. Calls which are updating a buffer which either might be used as a structure containing pointers, or where we could not figure out where the buffer escapes to.

   `memcpy`: 763 calls, `memset`: 542 calls.

The third and fourth categories are those where the `memcpy`/`memset` could affect future reads or writes of structure fields, but which we are not modelling. These include 2820 total calls to `memcpy` or `memset` within the kernel (27% of all such calls), one for every 32 functions in the kernel.

Not all of these calls are important to analyses; by examination, many appear to target memory used for doing device I/O, involving the same structures used in external casts (see Section 7.4). For example, 637 calls target the data buffers in a `buffer_head` (used for doing disk I/O, see Section 7.4.1) or an `sk_buff` (used for doing network interface I/O).

9.3.2 Trace aliasing

The rules we presented for local memory trace aliasing in Section 5.3.1 are rather coarse. For any two pointers `p1` and `p2` of the same type, `*p1` and `*p2` are considered aliased, as well as `p1->f` and `p2->f`.

At least for the applications for which we use the local memory analysis, this coarseness has not been a problem. During analysis, few aliasing pairs are relevant to either the locations used and updated in each function.

During the local memory analysis, `trace_alias` queries will be performed between any trace which is updated at a point in the function, and any trace whose value the analysis wants to know at the point after the update. The actual traces used for the latter are dependent on the

client analysis. There is, however, a large baseline set in all the traces which may be dereferenced later in the function.

Over the 125792 functions and loop bodies analyzed, the local memory analysis finds 79616 possible aliasing pairs of traces between this baseline set and the written traces, an average of .63 aliases per function/loop.

As with the calls to `memcpy` and `memset`, a large proportion of these aliases are irrelevant to our analyses and to most other program analyses. 43296 of the aliases, or 54%, are between elements in arrays of integers or characters.

20814 (26%) of the total aliases have a field trace `fld{_,_,_}` as one of the two aliased traces. Of these, only 384 aliases are between a field trace and a non-field trace — it is extremely rare that a field `x.f` could alias some pointer `*p` if both are used in the same function. The remaining field aliases are between traces with the same field but different base structures.

6926 of these field aliases are on fields with pointer type. These are the only places in the kernel, one for every 18 functions, where any `x.f` of pointer type could be found to alias another trace modified in that same function. None of these field aliases, nor any other aliases, have been too imprecise for later analyses, and we have not had to add any annotations to improve the behavior of the trace aliasing rules.

9.3.3 Function Purity

Out of a total of the 91384 functions, we have identified 17999 functions (20%) as semi-pure — we can exactly model the small set of locations the function might modify. This includes 68% of the functions which are not part of and do not call into the call graph SCC described in

Section 5.3.2.

Of the 17999 semi-pure functions, 13816 functions (15% of the total) are truly pure and do not have any side effects at all.

While a minority in the kernel's functions, these semi-pure and pure functions are called far more often in comparison to other kernel functions. Of all calls in the kernel, 30% are to one of the truly pure functions, and 36% are to a function that is either pure or semi-pure.

For the remaining 64% of calls, though, we have to clobber all locations that are not stack locals of the function and are not semi-pure fields of a heap structure.

9.3.4 Field Purity

Out of 15820 structure fields of pointer type written to in the kernel, we have identified 8874 semi-pure fields (56%), those written to only shortly after allocation. Out of 52395 fields of integer or character type written to, 24943 are semi-pure (48%). This large discrepancy gives the impression of a heap where much of the data is static and unchanging, in particular the pointer links between objects.

Still, our field purity analysis can get confused and mark fields that are really semi-pure as impure. The numbers above are underapproximations, and while the true numbers of such semi-pure fields may not be much greater, some of the most important pointer fields have the most complex initialization code, and for these we fall back on annotations.

Annotations

We use 25 annotations for fields of types that we believe are semi-pure, but for which we cannot prove this. Most of these are due to

initialization code involving tens to hundreds of functions and numerous indirect calls; at some point the escape analysis will fail to find the allocation sites for certain important structures being initialized. Of particular note are device probing and file creation, which together account for 19 of the 25 annotations. We give an example for device probing below.

In order to use an attached hardware device, the kernel must first probe the device; it has to identify the correct driver to interact with the device, and initialize various device-specific data structures for the driver to use. Some of these initialization functions are deep in the call chain during probing and have complex behavior, causing problems. An example is `snd_cmi8330_pcm`, which initializes parts of the `snd_cmi8330` structure used by the driver for certain C-Media sound card chips:

```
// sound/isa/cmi8330.c
static int __devinit snd_cmi8330_pcm(struct snd_card *card,
                                     struct snd_cmi8330 *chip)
{
   struct snd_pcm *pcm;
   const struct snd_pcm_ops *ops;
   int err;
   static snd_pcm_open_callback_t cmi_open_callbacks[2] = {
      snd_cmi8330_playback_open,
      snd_cmi8330_capture_open
   };

   ...

   /* SB16 */
   ops = snd_sb16dsp_get_pcm_ops(CMI_SB_STREAM);
   chip->streams[CMI_SB_STREAM].ops = *ops;
   chip->streams[CMI_SB_STREAM].open = ops->open;
   chip->streams[CMI_SB_STREAM].ops.open =
      cmi_open_callbacks[CMI_SB_STREAM];
   chip->streams[CMI_SB_STREAM].private_data = chip->sb;
```

```
   ...
}
```

Type `snd_pcm_ops` is a function pointer table used for the PCM sound abstraction layer. For each substream of the sound card chip, `snd_cmi8330_pcm` gets a table via `snd_sb16dsp_get_pcm_ops` (which returns the address of one of two global `snd_pcm_ops` tables), copies the table to the substream's `ops` field, and changes the `ops.open` callback. These updates to the substream's function pointer table are not likely to occur while the chip is actually in use.

Indeed, `snd_cmi8330_pcm` is only called while probing the sound card to which the `cmi8330` driver will attach. Several levels up the call chain, `card` and `chip` were created by a call to `snd_cmi8330_card_new`:

```
// sound/isa/cmi8330.c
static struct snd_card *snd_cmi8330_card_new(int dev)
{
   struct snd_card *card;
   struct snd_cmi8330 *acard;

   card = snd_card_new(index[dev], id[dev], THIS_MODULE,
                  sizeof(struct snd_cmi8330));
   if (card == NULL) {
      snd_printk(KERN_ERR PFX "could not get a new card\n");
      return NULL;
   }
   acard = card->private_data;
   acard->card = card;
   return card;
}
```

The chip `acard` is stored at `card->private_data` (this is actually pointing to a section of memory at the end of `card`), which will eventually be passed in to `snd_cmi8330_pcm` as `chip` — At entry to

`snd_cmi8330_pcm`, it is the case that `chip == card->private_data`. This use of `private_data` is what confounds the escape analysis when trying to figure out where `chip` came from; every function where `card` is passed could potentially modify `card->private_data`, changing the value of `chip` at entry to `snd_cmi8330_pcm`. There are hundreds of such functions where `card` could flow to, and as such we cannot prove within `snd_cmi8330_pcm` that `chip` was recently allocated and that the fields of `snd_pcm_ops` are semi-pure. `snd_pcm_ops` is an important structure for hundreds of casts, so we add an annotation marking its fields as semi-pure.

9.4 Escape Analysis

Since the escape analysis is completely demand driven, it is even harder to gauge its effectiveness directly than our other analyses. It can and does break down, for example if we try to use it with the `kernel_param` example from Section 9.2.1 to find the possible function pointer values it will find a useless set of thousands of entries.

There isn't much we can do to address this with annotations; if it is possible to get a suitable set of entries with the escape analysis, getting this set depends on the level of precision (values of `t_escape_precision` from Section 5.2). Analyses depending on this information, in particular the polymorphic data analysis, try the same escape queries at multiple levels of precision.

The main use for aliasing annotations is for allocator functions; if we don't include special treatment for the base allocators then following any value back to a dynamic allocation will then continue propagating through the allocator's internal structures. While this propagation is legitimate, we aren't interested in the allocator behavior and will assume

it behaves correctly (returns zeroed or uninitialized data), stopping propagation at these allocation calls and treating the return values as fresh locations.

9.5 Polymorphic Data Analysis

Out of the 11976 indirect call sites covered by the function pointer analysis, 7850 (66%) of these were identified by the `init_poly_data` pass as potentially involving 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 potential structural relationships were identified, and an additional 5939 potential relationships were added due to transitive correlations (see Section 6.5.2), for a total of 14769 potential relationships. Of these, 9601 are between structure fields, and 5168 are between the arguments to a function.

After running the `poly_data` analysis itself to find possible correlations, we find that correlations were successfully found for 10416 (71%) of the relationships, which includes 5750 (60%) of the structure field relationships and 4666 (90%) of the function argument relationships. Of the 7850 call sites with potential relationships, correlations were found for at least one of those relationships at 6883 (88%) of the sites.

Now, most of these relationships will not end up being useful to the casting analysis; either they don't capture a property relevant to polymorphism, or their correlations end up being too overapproximate. In Section 9.6 we will come back to this question of which relationships are useful. First, though, we go into the annotations required to get the polymorphic data results to perform as it does.

The polymorphic data analysis uses annotations more extensively

than any of the other analyses we present, requiring 177 annotations total. This is largely due to the variety of polymorphic structures in Linux and the complexity of their initialization. For a given polymorphic structure, our analysis tends to fail to generate a good enough set of correlations in three ways.

- The actual polymorphism encoded by the structure might not fit the model of structural relationships our analysis uses (a function pointer and data trace at a particular offset from the same structure). We can sometimes fit these cases so that we can capture the correlations generated by the structure, even if our analysis of the structure's internals is largely incomplete.
- If the polymorphism does fit into the structural relationship model, the way in which the structure is initialized might not fit the inference techniques our analysis uses to find correlations. This is usually a close fit, and we can use annotations to adjust the inference to exactly match the initialization.
- If both the polymorphism and initialization fit our model, overapproximation by other analyses, such as call site clobbering and escape imprecision, might lead to overly overapproximate correlations which can be handled by annotations.

We give examples for each of these categories of annotations in the following subsections. We cover the categories in the reverse order from above, going from the least complex to most complex cases.

9.5.1 Analysis Overapproximation

The same problems we had in analyzing complex initialization code for finding relatively pure fields are also responsible for a variety of

overapproximation issues in the polymorphic data analysis. When initialization of the different fields of a data structure is split across many functions, we need to have precise knowledge of which fields are uninitialized, NULL, or non-NULL at the various execution points in order to generate precise correlations.

For example, consider again the `file` creation example we initially discussed in Section 6.2.

```
// 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;
   }

   ...
}
```

The structural relationships we are interested in here are 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 here in `__dentry_open`, and the `private_data` in the indirect call `open`.

The polymorphic data analysis sees the `f_op` write in `__dentry_open` and no `private_data` write, and so will correlate all the possible values of `f_op->read` (all file `read` functions in existence) with the input value `f->private_data`. (Note that simply inlining the possible targets of `open` would not fix this issue; some of the `open` methods do **not** set `private_data`, as those target filesystems never use that field).

`__dentry_open` is only called during initialization of `f`, however, and the only possible value for `f->private_data` at entry to `__dentry_open` is `NULL`. We cannot detect this; if we could, the polymorphic data analysis would not add any correlations for the write in `__dentry_open`, as desired.

The reason we can't determine `f->private_data` is `NULL` is, again, tricky initialization code. Most of the time `__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);
}
```

Another caller of `__dentry_open`, `lookup_instantiate_filp`, passes in as the file the value `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 was allocated on the stack several levels up the call chain and across potentially multiple indirect calls. The `nameidata` is used throughout the lookup/create operation on a `file`, which can be very complex for some filesystems such as NFS.

9.5.2 Unhandled Initialization

Sometimes data structures have important structural relationships, but the way in which they are initialized is unusual enough to not at all fit the styles modelled by our inference algorithm. Some of the sound PCM layer structures we discussed briefly in Section 9.3.4 fit this model. The `snd_pcm_ops` are function pointer tables primarily used by the `snd_pcm` structure 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;
   /* -- substreams -- */
   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, and each `snd_pcm_str` has a list `substream` of `snd_pcm_substream` structures 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, which work in pretty much the same way as the `file` structural relationships from Section 6.2. However, writes to the `ops` and `private_data` fields of `snd_pcm_substream` are not correlated with one

another in the usual way. Instead of writing to 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 of this 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 this initialization code, the substream will have its `ops` set

but not its private_data. This state will persist until the substream is opened in snd_pcm_open_substream, which must be called before the substream can be used. snd_pcm_open_substream 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;
}
```

Thus, by correlating a write to the `snd_pcm->private_data` with calls to `snd_pcm_set_ops`, a PCM driver ensures that when the substream is eventually opened that the correlation between the value of `private_data` and the ops used in the `snd_pcm_set_ops` call will be introduced as correlations within the `snd_pcm_substream` structural relationship.

To annotate this case, we have to add custom rules to the polymorphic data analysis that 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.

9.5.3 Unhandled Polymorphism

The most interesting uses of polymorphism are those which our analysis is not even capable of expressing. There are not many instances of these, but they are generally big and important and responsible for many casts we will fail to prove later on (examples are in Section 9.6).

One of these cases, the Sysfs filesystem, we have managed to annotate, providing a mechanism to check the casts performed by clients of

Sysfs with the limitation that our annotations are, as usual, trusted; we assume that Sysfs follows the annotated behavior.

Sysfs is a Linux filesystem which provides a mechanism for users to query and update attributes of the drivers and associated devices that are currently mounted, by accessing files stored under the `/sys` directory. To the driver writer, this functionality is hidden behind a simple polymorphic interface, which relates a kernel object `kobj` (each device used in Sysfs will have 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 can use `sysfs_create_file` by passing in the kernel object for its device and the attribute it wants 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 such that the `disk` parameter to `aoedisk_show_state` will be equal to the `d->gd` value as passed into a call to `aoedisk_add_sysfs`. The question is how this happens, what machinery is hidden behind `sysfs_create_file` and the filesystem itself to ensure `aoedisk_show_state` is called with the right value.

`sysfs_create_file` (which we will go into more detail later) creates a file with the following `sysfs_file_operations` operations (see Section 6.2 for the earlier discussion of the `file_operations` structure).

```
// 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,
};
```

Whenever the user tries to read this file, the `sysfs_read_file` function will be called, which will invoke `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);
   ...
}

// 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,
};
```

`sysfs_read_file` calls the helper function `fill_read_buffer`, which gets an `attribute` 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. When the attribute being read is `disk_attr_state.attr` (or any other attribute of a `gendisk`), the `ops` will point to `disk_sysfs_ops`, and `ops->show` will call `disk_attr_show`. `disk_attr_show` 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`). `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. The main concept is that Sysfs will create a single directory for each device `kobject`, and a file within this directory for each `attribute` added for that `kobject`.

In order to specify these invariants in more detail, we first need to lay out the various involved structures and fields.

- `file`: A file usable for reading/writing (in general, multiple files may correspond to the same `inode`). Has fields `f_dentry` for the `dentry` used to open it, and `private_data` for filesystem private data.

- `dentry`: A directory entry, a particular path used to access some inode. Has fields `d_parent` for the parent `dentry`, (the directory which this was accessed from), and `d_fsdata` for filesystem private data.

- `sysfs_buffer`: A buffer used by Sysfs to read or write data to individual attributes. Each `file` in Sysfs has a `sysfs_buffer`

as its `private_data`. Has a field `ops` of type `sysfs_ops` with operations for reading/writing to it.

- `sysfs_dirent`: Per-`dentry` private data stored by Sysfs; each `dentry` that is not a symbolic link has a `sysfs_dirent` at its `d_fsdata`. Has a `void*` field `s_element` for additional data. When the `dentry` is for a directory, this field points to a `kobject`, and when the `dentry` is for a file, this field points to a `attribute`.

  The `s_element` fields are accessed through the following helpers by `fill_read_buffers` and other functions.

```
// 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);
}
```

  The last complication with `sysfs_dirent` is that this structure maintains a tree hierarchy which mirrors the filesystem hierarchy (minus any symbolic links). This tree is implemented with doubly linked lists; each `sysfs_dirent` has a `s_children` entry as the head of its list of children, and a `s_sibling` entry as its position in the list of its parent's children.

Now, the invariants maintained for our example are over three objects:

- `file`: The file accessed by `sysfs_read_file`.
- `kobject`: The kernel object for the device.
- `attribute`: The attribute for the `kobject` which can be queried or updated by accesses to `file`.

As specified in the descriptions above, for all Sysfs files and directories the private data pointers refer to particular types of objects. Moreover, the file `dentry` has a pointer to `attribute`, and the containing directory `dentry` has a pointer to `kobject`. `kobject` has a pointer back to this directory `dentry`. The only additional concern is making sure the `sysfs_ops` used by the `sysfs_buffer` in `file->private_data` points to whichever ops are suited for `kobject`; this will be a pointer either at `kobject->kset->ktype->sysfs_ops`, or `kobject->ktype->sysfs_ops`. For a `gendisk` kernel object, the `kobject->kset->ktype->sysfs_ops` points to `disk_sysfs_ops`.

The full set of data invariants required is as follows:

1. `file->f_dentry->d_parent->d_fsdata` is a `sysfs_dirent`
2. `file->f_dentry->d_parent->d_fsdata->s_element == kobject`
3. `file->f_dentry->d_parent == kobject->dentry`
4. `file->f_dentry->d_fsdata` is a `sysfs_dirent`
5. `file->f_dentry->d_fsdata->s_element == attribute`
6. `file->f_dentry->d_fsdata` is in the `s_children` list of children for `file->f_dentry->d_parent->d_fsdata`.
7. `file->private_data` is a `sysfs_buffer`

8. `file->private_data->ops` is either `kobject->ktype->sysfs_ops` or `kobject->kset->ktype->sysfs_ops`.

Establishing these invariants takes place in four function calls, which occur sequentially during execution before the `sysfs_read_file` can be called. We will go over each of these functions.

- Invariant 1,2,3: `sysfs_create_dir` makes the directory for `kobject`.
- Invariant 5,6: `sysfs_create_file` attaches an attribute to the `kobject`'s directory (despite the name, it does not actually create a `file` object).
- Invariant 4: `sysfs_lookup` attaches the `dentry` which will be used for `file` to the attribute.
- Invariant 7,8: `sysfs_open_file` sets up the file for reading/writing.

`sysfs_create_dir` is called for each `kobject` which will have attributes accessible via Sysfs.

```
// fs/sysfs/dir.c

int sysfs_create_dir(struct kobject * kobj)
{
   struct dentry * dentry = NULL;
   struct dentry * parent;
   ...

   create_dir(kobj,parent,kobject_name(kobj),&dentry);
   kobj->dentry = dentry;
}

static int create_dir(struct kobject * k, struct dentry * p,
                      const char * n, struct dentry ** d)
{
   ...
```

```
   *d = lookup_one_len(n, p, strlen(n));
   if (!sysfs_dirent_exist(p->d_fsdata, n))
      sysfs_make_dirent(p->d_fsdata, *d, k, mode,
                  SYSFS_DIR);
   ...
}

int sysfs_make_dirent(struct sysfs_dirent * parent_sd,
                      struct dentry * dentry,
                      void * element, umode_t mode, int type)
{
   struct sysfs_dirent * sd;
   sd = sysfs_new_dirent(parent_sd, element);

   sd->s_mode = mode;
   sd->s_type = type;
   sd->s_dentry = dentry;
   if (dentry) {
      dentry->d_fsdata = sysfs_get(sd);
      dentry->d_op = &sysfs_dentry_ops;
   }
}

static struct sysfs_dirent *
sysfs_new_dirent(struct sysfs_dirent * parent_sd,
                 void * element)
{
   struct sysfs_dirent * sd;
   sd = kmem_cache_alloc(sysfs_dir_cachep, GFP_KERNEL);

   memset(sd, 0, sizeof(*sd));
   atomic_set(&sd->s_count, 1);
   atomic_set(&sd->s_event, 0);
   INIT_LIST_HEAD(&sd->s_children);
   list_add(&sd->s_sibling, &parent_sd->s_children);
   sd->s_element = element;

   return sd;
}
```

```
// fs/sysfs/sysfs.h
static inline struct sysfs_dirent *
sysfs_get(struct sysfs_dirent * sd)
{
   if (sd) {
      atomic_inc(&sd->s_count);
   }
   return sd;
}
```

`sysfs_create_dir` takes the `kobject`, gets the parent directory entry (typically the filesystem root), and, within the helper `create_dir`, gets the new `dentry` at `*d` via `lookup_one_len`. `sysfs_make_dirent` creates a new `sysfs_dirent sd` such that `(*d)->d_fsdata == sd` and `sd->s_element == k` (invariants 1 and 2), and also adds `sd` to the list of the parent entry's children. `sysfs_create_dir` fills `kobject->dentry` on return from `create_dir` (invariant 3).

As shown in the initial example, `sysfs_create_file` is called for each attribute which is added for a `kobject`.

```
// fs/sysfs/file.c

int sysfs_create_file(struct kobject * kobj,
                      const struct attribute * attr)
{
   return sysfs_add_file(kobj->dentry, attr, SYSFS_KOBJ_ATTR);
}

int sysfs_add_file(struct dentry * dir,
                   const struct attribute * attr, int type)
{
   struct sysfs_dirent * parent_sd = dir->d_fsdata;
   umode_t mode = (attr->mode & S_IALLUGO) | S_IFREG;

   if (!sysfs_dirent_exist(parent_sd, attr->name))
      sysfs_make_dirent(parent_sd, NULL, (void *)attr,
```

```
                    mode, type);
}
```

`sysfs_create_file` just calls `sysfs_make_dirent` to, again, make the new `sysfs_dirent` `sd`, set its `s_element` to `attr` (invariant 5), and add `sd` onto the list of children of the `kobject->dentry->d_fsdata` `sysfs_dirent` (invariant 6).

The directory entry that will be used for `file` itself is not attached until `sysfs_lookup` is called.

```
// fs/sysfs/dir.c

static struct dentry * sysfs_lookup(struct inode *dir,
                                    struct dentry *dentry,
                                    struct nameidata *nd)
{
   struct sysfs_dirent * parent_sd = dentry->d_parent->d_fsdata;
   struct sysfs_dirent * sd;

   list_for_each_entry(sd, &parent_sd->s_children, s_sibling) {
      if (...) {
         ...
         sysfs_attach_attr(sd, dentry);
         break;
      }
   }
}

static int sysfs_attach_attr(struct sysfs_dirent * sd,
                             struct dentry * dentry)
{
   ...
   dentry->d_fsdata = sysfs_get(sd);
   sd->s_dentry = dentry;
   ...
}
```

`sysfs_lookup` takes the `dentry` for the file which the lookup is on,

gets the list of attributes for the directory the file is in, and looks for a matching `sysfs_dirent sd`, which it attaches to the `dentry` (invariant 4).

Finally, after lookup then `sysfs_open_file` will be called on the `file` to prepare it for reading/writing.

```
// fs/sysfs/file.c

static int sysfs_open_file(struct inode *inode, struct file *filp)
{
   return check_perm(inode,filp);
}

static int check_perm(struct inode *inode, struct file *file)
{
   struct kobject *kobj =
      sysfs_get_kobject(file->f_dentry->d_parent);
   struct sysfs_buffer * buffer;
   struct sysfs_ops * ops = NULL;
   ...

   if (kobj->kset && kobj->kset->ktype)
      ops = kobj->kset->ktype->sysfs_ops;
   else if (kobj->ktype)
      ops = kobj->ktype->sysfs_ops;
   ...

   buffer = kzalloc(sizeof(struct sysfs_buffer), GFP_KERNEL);
   if (buffer) {
      buffer->needs_read_fill = 1;
      buffer->ops = ops;
      file->private_data = buffer;
   }
   ...
}
```

`sysfs_open_file` allocates and fills in the `sysfs_buffer` for the file (invariant 7), setting the `ops` field to the `sysfs_ops` associated with the

parent directory's `kobject` (invariant 8). (`sysfs_get_kobject` functions as `to_kobj` from earlier, except with special handling for symbolic links.)

Thus, all the invariants required for the call to `sysfs_read_file` to invoke `aoedisk_show_state` correctly are established by prior calls setting up the filesystem structure.

9.6 Casting Analysis

The population of downcasts we are interested in proving the safety of are those casting to a structure type which transitively contains any pointer fields (for the rationale behind this, see Section 7.4.2). This includes a total of 28767 casts in the version of Linux we analyzed, including casts to 2723 different types. Of these casts we are able to prove 21637 as safe, 75.2% of the total. (As was described in Section 1.4, there are a few caveats to the word 'prove' here). This leaves 7130 casts as possibly unsafe.

Of the proved casts, 4014 (19%) are fixing the type of a location immediately after it is created by, e.g. `kmalloc`. The remainder are largely non-trivial to prove correct, owing in large part to the extensive use of structures. Points which directly cast the value of a structure field include 16673 (58%) of the total casts, 11615 (54%) of the casts we are able to prove, and 5058 (71%) of the casts we are not able to prove.

Moreover, these casts are concentrated in a relatively small number of fields. Only 580 fields have any casts at all on them. Of the 28767 total casts, 13717 (48%) are on a set of 51 fields with at least 50 casts each, and 10333 (36%) are on a set of 17 fields with at least 200 casts each.

For the missed casts, 215 fields have at least one missed cast. Of the 7130 missed casts, 3444 casts (48%) are on a set of 24 fields with at least 50 missed casts each. One of these fields is the `file->private_data` field; in total there are 970 casts of this field, of which we cannot prove 88.

As significant as the numbers of casts of fields are, they still exclude casts which are only indirectly involved with structures — the value being cast came from a structure but was copied through one or more function arguments before the actual cast. An example is the `timer_list` structure from Section 1.1, where `saa7146_buffer_timeout` casts its first argument after it was copied from a structure field by the parent function `__run_timers`.

We can get a better handle on these more indirect casts by looking at the population of casts we were able to prove where that proof depends on some structural relationship the polymorphic data analysis captured. This includes a total of 8754 (40%) of the proved casts. Of this group, 1755 casts are like `saa7146_buffer_timeout` and only indirectly depend on a structure field.

As with `void*` structure fields, there is a small group of structures with polymorphic relationships responsible for most of the proved casts (not surprisingly, these two sets have significant overlap). 173 different structures have some associated relationship we 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).

Understanding casts requires us to understand precisely how these

`void*` pointers flow through the heap, being stored by one function and eventually read by some other function far away in the call graph. Handling these data structures requires us to build analyses that both closely model the common case (most of the 173 polymorphic data structures) and allow us to use annotations to account for the quirks of the most important and frequently used structures.

In Section 9.6.1 we describe the `net_device`, a structure with some quirks that we are able to model very well and prove more casts on than any other structure. In Sections 9.6.2 and 9.6.3 we describe the `device` and `super_block` structures, for which this approach breaks down due to imprecision we cannot resolve with annotations. We miss more casts on these two structures than any other, and while these are still a small fraction of the missed casts, we feel they are somewhat representative of the remaining difficulties in analyzing complex code.

9.6.1 Net device allocation

In almost all cases structure allocation works as described in Section 7.2, where the primitive allocators are called with maybe a few wrappers and the result immediately cast to a new type. A few structures use a more ad hoc style of allocation, where a single dynamic allocation is used to store several structures of different types. This is uncommon but the cases where it is used are important, and the most important such case is the `net_device` structure, illustrated by the following example cast:

```
// drivers/net/bnx2.c
static int
bnx2_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
   struct bnx2 *bp = netdev_priv(dev);
   ...
```

```
   if (unlikely(bnx2_tx_avail(bp) < ...)) {
      ...
   }
   ...
}

// include/linux/netdevice.h

#define NETDEV_ALIGN         32
#define NETDEV_ALIGN_CONST     (NETDEV_ALIGN - 1)

static inline void *netdev_priv(struct net_device *dev)
{
   return (char *)dev + ((sizeof(struct net_device)
                                + NETDEV_ALIGN_CONST)
                         & ~NETDEV_ALIGN_CONST);
}
```

`bnx2_start_xmit` casts the result of `netdev_priv` and proceeds to use that value. However, `netdev_priv` does not access a field of the device, but advances the `dev` pointer past the end of its structure and returns that value.

This is safe to do because when `dev` was allocated a larger size was used than `sizeof(net_device)`. This allocation was performed by a call to `alloc_etherdev` in `bnx2_init_one` (note that `bnx2_init_one` sets the `hard_start_xmit` field of the device to `bnx2_start_xmit`, introducing a structural correlation so that `bnx2_start_xmit` will later be called with this same device).

```
// drivers/net/bnx2.c
static int __devinit
bnx2_init_one(struct pci_dev *pdev,
              const struct pci_device_id *ent)
{
   struct net_device *dev = NULL;
   ...
```

```
   dev = alloc_etherdev(sizeof(*bp));
   ...

   dev->open = bnx2_open;
   dev->hard_start_xmit = bnx2_start_xmit;
   dev->stop = bnx2_close;
   ...
}

// net/ethernet/eth.c
struct net_device *alloc_etherdev(int sizeof_priv)
{
   return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
}

// net/core/dev.c
struct net_device *alloc_netdev(int sizeof_priv, const char *name,
                                void (*setup)(struct net_device *))
{
   void *p;
   struct net_device *dev;
   int alloc_size;

   alloc_size = (sizeof(*dev) + NETDEV_ALIGN_CONST)
                & ~NETDEV_ALIGN_CONST;
   alloc_size += sizeof_priv + NETDEV_ALIGN_CONST;

   p = kzalloc(alloc_size, GFP_KERNEL);
   if (!p) {
      printk(KERN_ERR "alloc_dev: Unable to allocate device.\n");
      return NULL;
   }

   dev = (struct net_device *)
         (((long)p + NETDEV_ALIGN_CONST) & ~NETDEV_ALIGN_CONST);
   dev->padded = (char *)dev - (char *)p;

   if (sizeof_priv)
```

```
        dev->priv = netdev_priv(dev);

    setup(dev);
    strcpy(dev->name, name);
    return dev;
}
```

While the size of the private data area of the `net_device` is set by the `sizeof_priv` argument to `alloc_netdev`, we will not be able to determine what **type** that private data should be treated as until we get to the call in `bnx2_init_one`. We use a few annotations to look for calls to `alloc_etherdev` or `alloc_netdev` where the private data size is `sizeof(some_type)`, and then fix the type of the result's private data to that `some_type`. This is just a guess, though it is safe to guess here; if we pick the wrong type we will end up failing to prove casts later on.

Besides `net_device`, we have seen only a few other structures using an allocation scheme similar to this. Rather than build an analysis specifically for these allocators, it's best then to just annotate the allocation functions appropriately. `net_device` allocation requires 18 annotations in the `init_casting` pass (`alloc_netdev` has several other wrappers), but these annotations have a huge benefit. We are able to prove a total of 1391 casts using polymorphism in the `net_device` fields, all of which would fail without these annotations.

9.6.2 Device data

The most important field we are not able to prove most casts on is the private data of device drivers. In Linux each installed device has a `device` struct associated with it, as well as a `device_driver` which includes the function pointers used to interact with that device. The

`device` struct is pretty thin, with few fields other than those to support a device tree hierarchy and to store private data for the driver and its associated interfaces. The field we are most interested in is `driver_data`, which stores driver private data for the device.

```
// include/linux/device.h

struct device {
   struct klist        klist_children;
   struct klist_node   knode_parent;
   struct klist_node   knode_driver;
   struct klist_node   knode_bus;
   struct device     * parent;
   ...

   struct bus_type * bus;
   struct device_driver *driver;
   void      *driver_data;
   void      *platform_data;
   void      *firmware_data;
   ...
};

static inline void *
dev_get_drvdata (struct device *dev)
{
   return dev->driver_data;
}

static inline void
dev_set_drvdata (struct device *dev, void *data)
{
   dev->driver_data = data;
}

struct device_driver {
   const char        * name;
   struct bus_type    * bus;
```

```
   ...
   int    (*probe)    (struct device * dev);
   int    (*remove)   (struct device * dev);
   void   (*shutdown) (struct device * dev);
   int    (*suspend)  (struct device * dev, pm_message_t state);
   int    (*resume)   (struct device * dev);
};
```

There are 778 casts of the `driver_data` field (almost all done through the `dev_get_drvdata` method) which we are not able to prove. These are largely due to a technique for constructing and storing devices and drivers which is not apparent from the definitions of `device` and `device_driver` themselves. Instead of directly allocating a `device` and `device_driver` and including in these structures whatever data is needed for the driver to function (as in `file` and the other structures we have examined), the `device` and `device_driver` are embedded in outer structures representing a more specific *class* of devices, and it is in these wrapper classes where most of the work goes on.

An example of the use of these wrappers is for USB devices, which are responsible for about one third of the `driver_data` casts we miss. The wrappers for a USB device are `usb_interface` and `usb_driver`, which contain, respectively, `device` and `device_driver` structures as inner fields. (`usb_interface` structures are not one-to-one with the physical USB devices that are plugged in; a physical device can have multiple interfaces, in which case there will be multiple `usb_interface` devices for it).

```
// include/linux/usb.h

struct usb_interface {
   struct usb_host_interface *altsetting;
   struct usb_host_interface *cur_altsetting;
   unsigned num_altsetting;
```

```
   ...

   struct device dev;
   struct class_device *class_dev;
};

static inline void *usb_get_intfdata (struct usb_interface *intf)
{
   return dev_get_drvdata (&intf->dev);
}

static inline void usb_set_intfdata (struct usb_interface *intf,
                                     void *data)
{
   dev_set_drvdata(&intf->dev, data);
}

struct usb_driver {
   const char *name;

   int (*probe) (struct usb_interface *intf,
              const struct usb_device_id *id);
   void (*disconnect) (struct usb_interface *intf);
   ...

   struct device_driver driver;
   unsigned int no_dynamic_id:1;
};
```

The USB device driver methods `prove`, `disconnect` and so forth take a `usb_interface` directly, and principally work with that interface, rather than the `device` that is being wrapped. An example of such a driver is the driver for the Pegasus USB-to-ethernet adapter.

When the interface is disconnected (or at any other intermediate operation on the device), `pegasus_disconnect` is called where the inner device's `driver_data` points to a `pegasus` structure. This property was established by a call to `usb_set_intfdata` when the device was

originally probed. The `pegasus` struct is actually the private data of a `net_device` created by the driver, as described in Section 9.6.1.

```
// drivers/usb/net/pegasus.c

static void pegasus_disconnect(struct usb_interface *intf)
{
   struct pegasus *pegasus = usb_get_intfdata(intf);
   ...

   unregister_netdev(pegasus->net);
   ...
   free_netdev(pegasus->net);
}

static int pegasus_probe(struct usb_interface *intf,
                   const struct usb_device_id *id)
{
   struct usb_device *dev = interface_to_usbdev(intf);
   struct net_device *net;
   pegasus_t *pegasus;
   ...

   net = alloc_etherdev(sizeof(struct pegasus));
   if (!net) {
      dev_err(&intf->dev, "can't allocate %s\n", "device");
      goto out;
   }

   pegasus = netdev_priv(net);
   memset(pegasus, 0, sizeof (struct pegasus));
   ...
   pegasus->intf = intf;
   pegasus->usb = dev;
   pegasus->net = net;
   ...

   usb_set_intfdata(intf, pegasus);
   ...
}
```

```
static struct usb_driver pegasus_driver = {
   .name = driver_name,
   .probe = pegasus_probe,
   .disconnect = pegasus_disconnect,
   ...
};
```

After analyzing the wrapper which calls `pegasus_disconnect` (omitted), we can determine that `pegasus_disconnect` is only called when it is equal to this expression:

```
container_of(intf->dev.driver,usb_driver,driver)->disconnect
```

The case is similar with `pegaus_probe`. We can thus model the transfer of data from `pegaus_probe` to `pegasus_disconnect` with structural relationships between the `intf->dev.driver_data` and these two function pointers.

The problem is in figuring out the possible correlations for these relationships. These relationships refer to the value of the `driver_data` field, of course, but this field is used and written by drivers for many other non-USB devices, and if we can't prove these other writes can't affect these relationships, we will end up polluting the structural correlations with values for `driver_data` that cannot actually be passed to `pegasus_disconnect`, and cannot prove the cast.

There are several dozen such polluting writes, with an example as `hp100_eisa_probe`.

```
static int __init hgafb_probe(struct device *device)
{
   struct fb_info *info;
   ...

   info = framebuffer_alloc(0, NULL);
```

```
   if (!info) {
      iounmap(hga_vram);
      return -ENOMEM;
   }

   ...
   dev_set_drvdata(device, info);
   return 0;
}
```

There is no indication from this function or its callers whether the device passed to it might point to the same `intf->dev` as is used by `pegasus_disconnect`, so we will treat the data in `pegagus_disconnect` as possibly having type `fb_info`. It would be possible to build a logic that could distinguish the different classes of devices from one another at the points where they are probed, but this logic would have limited applicability; the `device` is the only place in Linux (or anywhere else) we have seen this coding pattern.

9.6.3 Filesystem superblocks

Another important field that we are not able to handle is the private data field of a filesystem `super_block`. Each mounted filesystem in Linux is associated with a superblock, a structure that holds data about both the filesystem's in-memory status (active and dirty inodes and files, etc.) and its on-disk structure and metadata (block sizes, flags, etc.). The superblock contains a field `s_fs_info` for storing data private to the particular filesystem that is mounted.

```
// include/linux/fs.h
struct super_block {
   ...
   struct super_operations *s_op;
   ...
```

```
   struct list_head         s_inodes;
   struct list_head         s_dirty;
   ...
   void                    *s_fs_info;
   ...
};
```

There are 561 casts of the `s_fs_info` field which we are not able to prove. One of these is in the `EXT2_SB` function below, which accesses the private data of a superblock for an ext2 filesystem (this wrapper is called in 131 places, though we only count it as a single cast, in effect artificially lowering the number of casts we end up finding for the `s_fs_info` field; there aren't many wrapper functions like this).

```
// include/linux/ext2_fs.h
static inline struct ext2_sb_info *EXT2_SB(struct super_block *sb)
{
   return sb->s_fs_info;
}

// include/linux/ext2_fs_sb.h
struct ext2_sb_info {
   ...
   struct buffer_head ** s_group_desc;
   unsigned long  s_mount_opt;
   uid_t s_resuid;
   gid_t s_resgid;
   ...
};
```

We fail to prove The `EXT2_SB` cast and other casts of `s_fs_info` largeley because the superblock is not accessed 'directly' through a method on its `super_operations` but indirectly through methods on the `inode` structures it contains. Each `inode` contain in-memory status for a file or directory, independent of the path name used to reach that file/directory, and for each file/directory currently in use for the

mount, there is an `inode` allocated and stored in the `s_inodes` list of that mount's `super_block`.

```
// include/linux/fs.h

struct inode {
   ...
   struct list_head i_sb_list;
   ...
   struct inode_operations *i_op;
   const struct file_operations *i_fop;
   struct super_block     *i_sb;
   ...
};

struct inode_operations {
   ...
   int (*mkdir) (struct inode *,struct dentry *,int);
   int (*rmdir) (struct inode *,struct dentry *);
   ...
   int (*setattr) (struct dentry *, struct iattr *);
   int (*getattr) (struct vfsmount *mnt, struct dentry *,
                   struct kstat *);
   ...
};
```

An example use of `EXT2_SB` when accessed through an inode is in `ext2_acl_chmod`, which checks the `s_mount_opt` flags of the an inode's superblock before modifying the access control list for that inode. `ext2_acl_chmod` is called indirectly by `notify_change` via the `inode->setattr` method.

```
// fs/ext2/acl.c
int
ext2_acl_chmod(struct inode *inode)
{
   struct posix_acl *acl, *clone;
   int error;
```

```
   if (!(EXT2_SB(inode->i_sb)->s_mount_opt & EXT2_MOUNT_POSIX_ACL))
      return 0;
   if (S_ISLNK(inode->i_mode))
      return -EOPNOTSUPP;
   ...
}

// fs/ext2/inode.c
int ext2_setattr(struct dentry *dentry, struct iattr *iattr)
{
   struct inode *inode = dentry->d_inode;
   int error;

   ...
   error = inode_setattr(inode, iattr);
   if (!error && (iattr->ia_valid & ATTR_MODE))
      error = ext2_acl_chmod(inode);
   return error;
}

// fs/attr.c
int notify_change(struct dentry * dentry, struct iattr * attr)
{
   struct inode *inode = dentry->d_inode;
   ...

   if (inode->i_op && inode->i_op->setattr) {
      error = security_inode_setattr(dentry, attr);
      if (!error)
         error = inode->i_op->setattr(dentry, attr);
   }
   ...
}

// fs/ext2/namei.c
struct inode_operations ext2_dir_inode_operations = {
   ...
   .mkdir        = ext2_mkdir,
```

```
    .rmdir        = ext2_rmdir,
    ...
    .setattr      = ext2_setattr,
    .permission   = ext2_permission,
};
```

We can represent the expectation within the `setattr` method about the type of the superblock's private data as a structural relationship on `inode` between `.i_sb->s_fs_info` and `.i_op->setattr`, and need to look for writes to the `i_sb`, `s_fs_info`, `i_op` and `setattr` fields which may introduce correlations for this relationship. All of these but `i_op` can be handled fairly easily. `i_sb` and `s_fs_info` are only written very shortly after the `inode` and `super_block` are allocated, respectively, and `setattr` is only written during static initialization.

Understanding the writes to `i_op` requires some highly specialized reasoning which transcends many functions, similar to the `device` case and unlike the other specialized structures we have described such as `net_device` and the Sysfs `attribute`. In-memory `inode` structures can be created in two main ways by filesystems: either an existing on-disk inode is opened and the in-memory structure is created for that on-disk structure, or a new file or directory inode is created, which will create first the in-memory `inode` and then later write out the on-disk one.

For ext2, the former is done through `ext2_read_inode`. If the `iget` function fails to find an in-memory inode for an inode number `ino` in the superblock, it will create a new `inode` via `alloc_inode`, which sets the `i_sb` to the superblock passed to `iget` and sets the `i_ops` to the `empty_iops` empty `inode_operations`. `iget` then calls `ext2_read_inode` through the superblock's operations to fill in the `i_op` and other fields with their final values.

```
// fs/ext2/inode.c
```

```
void ext2_read_inode (struct inode * inode)
{
   ...
   if (S_ISREG(inode->i_mode)) {
      inode->i_op = &ext2_file_inode_operations;
      ...
   } else if (S_ISDIR(inode->i_mode)) {
      inode->i_op = &ext2_dir_inode_operations;
      ...
   }
   else if (S_ISLNK(inode->i_mode)) {
      ...
   }
   ...
}

// include/linux/fs.h
static inline struct inode *
iget(struct super_block *sb, unsigned long ino)
{
   struct inode *inode = iget_locked(sb, ino);

   if (inode && (inode->i_state & I_NEW)) {
      sb->s_op->read_inode(inode);
      unlock_new_inode(inode);
   }

   return inode;
}

// fs/inode.c

struct inode *iget_locked(struct super_block *sb, unsigned long ino)
{
   struct hlist_head *head = inode_hashtable + hash(sb, ino);
   struct inode *inode;

   inode = ifind_fast(sb, head, ino);
   if (inode)
```

```
      return inode;

   return get_new_inode_fast(sb, head, ino);
}

static struct inode *
get_new_inode_fast(struct super_block *sb, struct hlist_head *head,
                   unsigned long ino)
{
   struct inode * inode;

   inode = alloc_inode(sb);
   if (inode) {
      ...
   }
   return inode;
}

static struct inode *alloc_inode(struct super_block *sb)
{
   static struct address_space_operations empty_aops;
   static struct inode_operations empty_iops;
   static const struct file_operations empty_fops;
   struct inode *inode;

   if (sb->s_op->alloc_inode)
      inode = sb->s_op->alloc_inode(sb);
   else
      inode = (struct inode *)
              kmem_cache_alloc(inode_cachep, SLAB_KERNEL);

   if (inode) {
      ...

      inode->i_sb = sb;
      ...
      inode->i_op = &empty_iops;
      inode->i_fop = &empty_fops;
      ...
```

```
   }

   return inode;
}
```

If we know that `ext2_read_inode` and other `read_inode` functions are always called through the pointer `inode->i_sb->s_op->read_inode` (we need an annotation for this, as we can't model the assignment of `i_sb` by `alloc_inode` within `iget` itself), then to get the correlations between `s_fs_data` and `i_op` we just need the possible correlations between `.s_op->read_inode` and `.s_fs_data` within `superblock`, which we will be able to infer with the polymorphic data analysis as usual.

The second case for `inode` creation does not directly involve the superblock. New file and directory inodes are created through the `inode_operations` of the directory inode which will contain them, and the `inode_operations` methods for that directory inode will copy its `i_sb` pointer to the `i_sb` of the newly created inode.

```
// fs/namei.c
int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
   ...
   error = dir->i_op->mkdir(dir, dentry, mode);
   if (!error)
      fsnotify_mkdir(dir, dentry);
   return error;
}

// fs/ext2/namei.c
static int ext2_mkdir(struct inode * dir, struct dentry * dentry,
                      int mode)
{
   struct inode * inode;
   int err = -EMLINK;
```

```
   if (dir->i_nlink >= EXT2_LINK_MAX)
      goto out;

   inode_inc_link_count(dir);

   inode = ext2_new_inode (dir, S_IFDIR | mode);
   ...

   inode->i_op = &ext2_dir_inode_operations;
   inode->i_fop = &ext2_dir_operations;
   ...
}

// fs/ext2/ialloc.c
struct inode *ext2_new_inode(struct inode *dir, int mode)
{
   struct super_block *sb;
   ...
   struct inode * inode;
   ...

   sb = dir->i_sb;
   inode = new_inode(sb);
   if (!inode)
      return ERR_PTR(-ENOMEM);

   ...
   return inode;
}

// fs/inode.c
struct inode *new_inode(struct super_block *sb)
{
   struct inode *inode;

   ...
   inode = alloc_inode(sb);
   if (inode) {
      ...
```

```
      list_add(&inode->i_sb_list, &sb->s_inodes);
      ...
   }
   return inode;
}
```

The relationship between the various `inode_operations` fields of an `inode` and the `i_sb->s_fs_info` are in effect mutually dependent, where the value for `setattr` depends on the value for `mkdir` and on up the directory tree to where the inode was read from disk. This mutual dependency is not by itself a problem; `file` and other structures have similar dependencies, where the value read by `read` is that which was written by `open`.

The difference is that for `inode_operations` the inodes involved are different from one another, and to follow the dependencies we need to show they share the same `i_sb` superblock. Proving **this** fact is very difficult; while we could annotate `alloc_inode` and `new_inode` and maybe some other special cases as filling in the `i_sb`, we also need to account for all their wrappers going back up to the `mkdir` or similar indirect call. These additional functions include `ext2_new_inode` and several dozen other functions.

As with the `device` case, the problem here is not one of the expressiveness of the summary information for capturing the relationships we need, but rather requiring too many annotations to ensure we get a sufficiently precise inference of those summaries.

Chapter 10

Related Work

C and C++ are alone among the widely used languages today in not providing type safety guarantees. Consequently, research has gone into ensuring either that C programs are type safe, and in replacing C with similarly expressive type safe alternatives. Most of this 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).

10.1 Checking type safety in C

Siff et. al. [25] 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 [25] 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. [19] 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 overhead of greater than 20x the original program's runtime.

HAVOC [18] is a static analysis system for C programs, which 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 small 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, in order to completely verify 5000 lines of code, HAVOC required 35 changes to the code, 36 trusted annotations (annotations which, like our annotations, will not be checked for correctness), and 153 untrusted annotations (those which will be 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.

10.2 C extensions for checking type safety

CCured [7, 21] 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 [21] 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.

Subsequent improvements [7] ensured that most pointers which are downcast are not fully `WILD`, but instead have 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 which uses a more lightweight approach than CCured, inserting runtime assertions where necessary but without changing the in-memory layout of pointers and other structures. When dealing with downcasts from one type to another, Deputy can soundly check 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 [3], 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 9.5.1).

Cyclone [12] 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 [2]; downcasts are disallowed. Types in Cyclone can be polymorphic [11] 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 will break on initializers such as the `file` open example (Section 9.5.1).

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 lets us 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.

10.3 Logic programming based analysis

Datalog and other logic programming languages have repeatedly been used for expressing program analyses [8, 10, 23, 30].

A challenge with Datalog is that while it is great for problems based on graph and set abstractions, it is a poor tool for modelling other abstractions such as boolean constraints. The previous applications of Datalog in program analysis have focused on problems expressed using sets and graphs, in particular pointer analysis [10, 30]. We are able to use more complex abstractions using predicate-based interfaces for calling into external solvers and other code (Section 2.3).

Constraint logic programming languages [15] fold constraints into

the language itself, such that the truth of a predicate instance depends on the satisfiability of a set of constraints. We do not go this far, as the great majority of predicates we construct do not depend on constraints. For those predicates involved with constraints, we can get a similar behavior (if not brevity) to a constraint logic programming language with extra constraint arguments to predicates (e.g. `guard` and `val` from Section 3.3).

10.4 Escape analysis

The information we need from our escape analysis is largely covered by the `timer_list` example from Section 5.1. Can we track the points to which a location might flow through a series of assignments, without conflating that location with other locations flowing to the same points? There is a great deal of work in this area.

Traditional escape analysis algorithms focus on dynamic allocations, and are primarily concerned with which call frames new locations might escape from (if data can't escape the frame it is allocated in, it can be stack allocated). These analyses originated with Ruggieri and Murtagh [24], and have largely been applied to garbage collected languages for use in optimization, including both functional languages [22] and Java [9]. These analyses can be quite precise intraprocedurally but provide very little information when a value **does** escape the function which allocated it.

Points-to analysis performs a global rather than per-function analysis of the target program and can generate the extra information we need in cases where a location escapes a function. Points-to analyses fall into two general categories, either unification based, as originally described by Steensgaard [27], or inclusion based, as originally described

by Andersen [4].

Unification based analyses unify the locations referred to by the source and target of all assignments in the program. In the context of the `timer_list` example, this will unify each `timer_list` in the program with the `timer_list` manipulated by `__run_timers`, and hence transitively unify each `timer_list` with every other `timer_list`, which is unacceptably imprecise.

Inclusion based analyses give the effect of only propagating information in assignments in one direction, and will let us distinguish the different `timer_list` structures from one another. Banshee [16, 17], a system for building constraint based analyses, has been used to build inclusion based analyses that scale to Linux. This analysis is *field-insensitive*, though; it ignores any distinctions between the different fields of a structure, which is again unacceptably imprecise.

Heintze and Tardieu describe an inclusion based analysis that incorporates fields and scales to programs the size of Linux [14]. This algorithm is completely demand driven, but has the chief drawback (from our perpsective) that it flattens all structures of the same type together — the different fields of a `timer_list` can be distinguished, but not different declarations of structures of type `timer_list`. If our escape analysis is run following forward edges at its minimal level of precision, it will compute pretty much the same information as [14]. In many (even most) cases this is all we need, but there also many cases that demand more precise results.

Sridharan et. al. give a similar demand driven analysis for program slicing that runs backwards rather than forwards and can distinguish different instances of the same type [26]. However, when indirection is used in the program a points-to analysis is needed to continue propagation (e.g. while tracking `y` and seeing `x.f := y`, the analysis propagates

to all `w.f` such that `w` may alias `x`). Reliance on such a points-to analysis again introduces the problem of getting such an analysis to scale up at a sufficient level of precision.

What allows us to use a purely demand driven algorithm as in [14] while still handling multiple levels of indirection is being able to follow assignment edges both backwards and forwards when there **is** indirection. This will exhaust the space of points the location could flow to or have flowed from, without falling back on eagerly computed unification or inclusion points-to information.

10.5 Memory analysis

The local memory model (Section 3.3) that forms the basis of most of our later analyses is based on that in the original version of Saturn by Xie and Aiken [31]. This approach of using a path-sensitive analysis intraprocedurally while aggressively summarizing at function boundaries is currently unrivalled in its ability to scale to arbitrarily large programs while retaining high function-level precision.

The changes we have made to this memory model are primarily to perform sound analysis of functions while still retaining the needed scalability and precision.

- Instead of analyzing loops by unrolling them a bounded number of times, we convert loops to tail recursive functions and fixpoint their summaries, computing all possible behaviors of the loop.
- Instead of treating all incoming locations as non-aliased, we account for possible aliasing using the involved fields, pointer types, and escape information for the locations.

- Instead of treating function calls as modifying nothing, we conservatively clobber locations if we cannot prove they won't be clobbered due to the modifies set of the function being called or fields in the location.

In previous work [13] we found that aliasing is used only to a limited extent in systems software. This has continued to be our experience evaluating the casting analysis (Section 9.3.2), with the exception of highly specialized data structures such as the Sysfs filesystem control structures (Section 9.5.3).

Characterizing modifies information on functions is, in our experience, a far harder problem, though our function- and field-based approach has worked well enough for the casting analysis. Our semi-pure fields are very similar to the stationary fields described by Unkel and Lam [29]. Our definition and inference is less restrictive and allows reads at any point on a semi-pure field, and for semi-pure fields to be written after heap objects have been updated to point to the containing structure. Our findings that roughly half of the fields in Linux are semi-pure are consistent with the experience of [29].

Chapter 11

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, not just whether a given cast is correct but how component interfaces and control structures in Linux are designed.

Bibliography

[1] *Berkeley DB*. http://www.oracle.com/database/berkeley-db/index.html.

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

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

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[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] Steven Dawson, C.R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Programming Language Design and Implementation*, 1996.

[9] Jong deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Object-Oriented Programming Systems, Languages and Applications*, 1999.

[10] Saha Diptikalyan and C.R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*, 2005.

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

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

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

[14] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *Programming Language Design and Implementation*, 2001.

[15] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20, 1994.

[16] John Kodumal. *Program Analysis with Regularly Annotated Constraints*. PhD thesis, 2006.

[17] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symposium*, 2005.

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

[19] 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.

[20] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction*, 2002.

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

[22] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Program Language Design and Implementation*, 1992.

[23] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, 1994.

[24] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Principles of programming languages*, 1988.

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

[26] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Programming Language Design and Implementation*, 2007.

[27] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, 1996.

[28] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[29] Christopher Unkel and Monica Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *Principles of Programming Languages*, 2008.

[30] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation*, 2004.

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