COMBINING HIGH LEVEL ALIAS ANALYSIS WITH LOW LEVEL CODE COMPACTION OF THE LINUX KERNEL

BY

JOHN EDISON TRIMBLE

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

DECEMBER 2006

Approved by:

Dr. Saumya Debray
Department of Computer Science

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for a degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Signed:________________________________

Contents

Abstract

The limited resources of embedded devices make it both costly and difficult to deploy with general-purpose operating systems such as Linux. The use of low level code compaction techniques can reduce the code size of the kernel to better suit the environment of an embedded device. However, using low level code compaction has typically precluded the use of high level aliasing information. This makes it difficult to resolve potential targets of indirect calls and which in turn reduces the degree of code compaction. This honors thesis discusses a method for combining high level aliasing information with low level code compaction of the Linux kernel by having the high level alias analysis construct part of the kernel call graph.

1 Introduction

As time goes on the use of embedded devices – such as cell phones, PDAs, MP3 players and the like – continues to grow. When choosing an operating system for an embedded device, vendors must decide between developing a specialized operating system in-house, or using a general purpose operating system such as Linux. Developing a home brew operating system is both a time consuming and expensive endeavor which deters such a pursuit. Unfortunately, general purpose operating systems such as Linux tend to be less sensitive to the limited resources of an embedded device, in particular the small amount of memory available. While the Linux kernel can be configured to remove unnecessary drivers, there are limits to the degree of custom tailoring that can be done to the kernel by simply changing the configuration.

Code compaction through binary rewriting has proven an effective means of reducing the code size of an application [5]. This requires – in part – the use of inter-procedural analysis to determine what producers will not be executed in a given application. This is complicated by the presence of indirect calls which make construction of the call graph problematic. This issue is compounded by both the lack of semantic information available in the decompiled binary and the large presence of indirect calls in the Linux kernel unlike traditional applications.

In the most conservative analysis, an indirect call could call any procedure in a given application. This, of course, makes the elimination of entire procedures from the kernel impossible though it does not preclude other types of code compaction. Another approach that is more suitable at the binary level, is to, at every indirect call, pose the question "which functions can have their addresses taken at this point in the programs execution?" This approach will mean for a given indirect call its targets will at most be any function in the application that has its address taken at some point. This is still a highly conservative approach but will at least provide *some* opportunity for procedure elimination. In order to achieve greater precision alias analysis must be used, which attempts to solve the more general problem of what pointers point to.

2 Background

The question of aliasing comes in two varieties. The first, and weakest is for *some* execution path P do two memory references a and b refer to the same memory location, this is called *may*-alias. The second and stronger assertion is that for *all* execution paths do two memory references a and b refer to the same memory location, this is known as *must*-alias. Unfortunately, both of these problems are, in general, undecidable [11]. This necessarily means that any alias analysis algorithm is an approximation algorithm. We can characterize these algorithms on several axis, most notably: flow-sensitivity, context-sensitivity, and type-sensitivity. For a detailed survey and discussion of alias analyses and their properties see [10].

Flow-sensitive alias analyses account for flow control within a procedure. They produce separate aliasing information for different points in the control flow of a procedure. Flow-insensitive analyses, by contrast, only produce aliasing information for the whole procedure and/or the whole program disregarding both flow control and statement execution order. Flow-insensitive analyses tend to be more conservative in their approximations but also less expensive than flow-sensitive analyses. Context-sensitive alias analyses distinguish between different invocations of a procedure. For example, if a procedure P is called from procedure Y and procedure Z, a context-sensitive analysis will produce aliasing information for both P_Y and P_Z. Context-insensitive analyses make no such distinction. The trade off between context-sensitive and context-insensitive analyses is the same as with flow-sensitive and flow-insensitive: more precision at a higher expense or less precision at a lower expense.

Type-sensitivity refers to using type information to assist in deriving aliasing information. For example, a simple type-sensitive analysis might infer that if references a and b have compatible types they may alias and otherwise they do not alias. Type-sensitivity tends to be cheap in terms of runtime and memory use though it does not – by itself – produce precise results. Alias analyses that are type-sensitive are generally limited to operating at the source level or some other high level program representation where type information is available.

Aside from issues of context-sensitivity, flow-sensitivity and the like, there is the issue as to what level of representation of a program does one perform alias analysis. When a program is written in several different languages, contains in line assembly, or does not have the complete source code available, perfroming alias analysis at a low level, such as on the machine code, is the apparent choice [6]. However, this comes at a cost as the loss of high level semantic information, such as type information, precludes an analysis from leveraging these semantics to improve aliasing precision such as through a type-sensitive alias analysis.

The FA analysis [14, 15] demonstrates well the effectiveness of performing alias analysis at the source level. The FA analysis is a flow-insensitive, context-insensitive, and type-sensitive alias analysis. Being both flow- and context-insensitive means that the algorithm has both a low cost, it runs in near linear time, and a low precision. Nonetheless, by utilizing type information in distinguishing between different fields of a structure, it is able to achieve surprisingly accurate results, especially in terms of call graph construction [12]. In fact, according to Milanova et al., the FA analysis produced the most accurate call graph possible in all of the tests they performed [12].

Clearly, source level analyses provide better results in terms of precision vs. cost – especially for call graph construction – because of their ability to take advantage of source level semantic information. The problem our research addresses is how to combine source code level alias analysis with machine code level code compaction and show that such a combination can produce a higher level of code compaction than performing the alias analysis at the machine code level.

3 FA Analysis

Here we will give a brief summary of the FA analysis developed by Zhang et al. [15]. As previously mentioned, the FA analysis is a flow-insensitive, context-insensitive, type-sensitive alias analysis. This particular analysis categorizes memory references into sets such that all the members in a given set may-alias one another. This has the side effect that if A may point to B then B may point to A. In other words, the FA analysis is symmetric.

The FA analysis does not take into account pointer arithmetic or array indexing. So a statement such as `*(p+1)` or `p[1]` is simply treated as `*p`. The analysis does, however, distinguish between different fields of a structure, so `p.x` is distinct from `p.y`.

The FA analysis constructs for each memory reference an object name. An object name is simply a variable name with a series of right-associative pointer dereferences (`*`) and address operators (`&`) as well as left-associative field accesses (`.field`). This construction mimics the use of memory references in C. For example, the reference in C `&(t->y)` would map to the object name `&(*(t).y)`. Since pointer arithmetic is ignored by the FA analysis the reference `*(p+1)` maps to the object name `*p`.

The PE equivalence relation (Pointer-related Equality) is used to partition the set of all object names into equivalence classes. This relation is represented by a graph G_{PE}. Each vertex of the graph corresponds to an equivalence class of object names. These vertices are connected via edges labeled as pointer-dereference or by a field name.

We will only provide a rough sketch of the construction of G_{PE}. The basic outline is to initially place every object name in its own equivalence class. Edges connect nodes as follows:

- If an equivalence class e_1 contains an object name o.`field` where o is an object name in equivalence class e_2, then there is an edge labeled `field` from the vertex v representing e_2 to the vertex w representing e_1 in G_{PE}.

- If an equivalence class e_1 contains an object name $*o$ where o is an object name in equivalence class e_2, then there is an edge labeled `*` from the vertex v representing e_2 to the vertex w representing e_1 in G_{PE}.

- If an equivalence class e_1 contains an object name &o where o is an object name in equivalence class e_2, then there is an edge labeled `*` from the vertex v representing e_1 to the vertex w representing e_2 in G_{PE}.

For every assignment in the given program the equivalence class of the object name representing left hand side of the assignment is merged with the equivalence class of the object name of the right hand side. This involves identifying the vertices representing these equivalence classes. If a merge results in a vertex v being the source of two edges with the same label, then the equivalence classes represented by the destination of these two edges are also merged.

Direct calls are handled similarly to assignments. The object names for the function's formal parameters are merged with object names for the parameters of the function call. Indirect calls are handled somewhat differently. Suppose an indirect call is made using function pointer p. Whenever a function's object name is merged with p then the object names for the call site's parameters are merged with the function's formal parameters.

After the G_{PE} is constructed, determining the potential targets of an indirect call is straight forward. If an indirect call uses a function pointer f then the possible targets are all the function object names in the same equivalence class as f.

4 Overview

Our solution to combining source level aliasing information with low level code compaction, is to implement a high level alias analysis – we will use the FA analysis – run this analysis on the source code, and then construct a partial call graph containing the call graph edges for indirect calls. The low level code compaction can then use this partial callgraph to construct the actual callgraph of the program by filling in the the remaining edges that are either normal function calls or an edge needed for some unrelated reason.

Figure 1 provides the general outline of the overall system. The system can be thought of as two chains, one performing the code compaction and another performing source level alias analysis. We will not discuss the code compaction chain in any great detail, it is mainly present for context. To demonstrate each phase of our system we will use a small example program Hello (Figure 2) which simply prints "hello world" via an indirect call.

In order to perform alias analysis of a given program, the source must first be parsed. For this task we use a GEM (GCC Extension Modules) plugin to hook into GCC's frontend. GEM was developed by Dr. Chiueh et al. at State University of New York and is simply a light weight patch for GCC that allows one to write extension modules for GCC to modify GCC's syntax trees and other intermediate representations at various phases of compilation [4]. Unfortunately, since compilation of an application generally occurs through multiple invocations of GCC – building multiple object files which are then linked – and our analysis depends upon information of the entire program. As a result, we do not attempt to perform the actual alias analysis inside our GCC extension

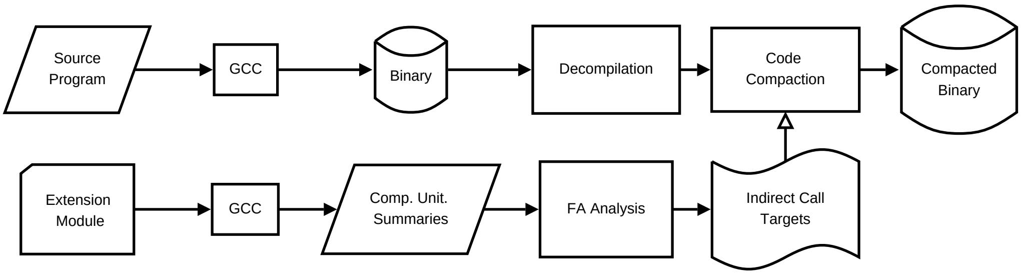

Figure 1: System Diagram

```
void hello(){ printf("hello world\n"); }

void main(void){
  (void *f)();
  f = &hello;
  f();
}
```

Figure 2: Hello program code.

module, instead we dump a summary for each compilation unit. Figure 3 shows the compilation unit summary for Hello.

The compilation unit summaries, once produced, are passed to a python script which implements the FA analysis algorithm. This script produces a partial call graph containing the edges for indirect calls. The call graph is written to a file named `callgraph`. Figure 4 shows the partial callgraph file produced for Hello.

5 Technical Details

5.1 GCC and GEM

One of the goals of our framework is to keep it as general as possible, since alias analysis is not specific to a particular piece of software or a particular platform our framework tries to embrace this generality. While GCC does work well with many different pieces of software, there is still a great deal of software which is not compatible with GCC and many pieces of software are only compatible with particular versions of GCC. For example, the 2.4 version of the Linux kernel will not compile with GCC 4 without a patch. Given this, the bulk of the analysis is moved outside of GCC itself.

GEM is a GCC kernel patch that allows one to define their own extension modules for GCC. The modules are in the form of a shared object file that defines a `gem_create` and a `gem_destroy` function which are called by GCC

```
<compunit>
  <function id="hello">
    <decl id="hello">
      <namespace/>
      <type-ref key="1"/>
    </decl>
    <functioncall>
      <decl id="printf">
        <namespace/>
      </decl>
      <param num="1">
        <decl id="@cststring1">
          <namespace/>
          <type-ref key="2"/>
        </decl>
      </param>
    </functioncall>
  </function>
  <function id="main">
    <decl id="main">
      <namespace/>
      <type-ref key="1"/>
    </decl>
    <assignment>
      <decl id="f">
        <namespace>
          <decl id="main">
            <namespace/>
            <type-ref key="1"/>
          </decl>
        </namespace>
      </decl>
      <address>
        <decl id="hello">
          <namespace/>
          <type-ref key="1"/>
        </decl>
      </address>
    </assignment>
    <functioncall>
      <decl id="f">
        <namespace>
          <decl id="main">
            <namespace/>
            <type-ref key="1"/>
          </decl>
        </namespace>
      </decl>
    </functioncall>
  </function>
  <type-table>
                ⋮
  </type-table>
<compunit>
```

Figure 3: Hello compilation unit summary.

```
main:hello,
```

Figure 4: Hello partial call graph.

at the beginning and end respectively of processing a compilation unit. GEM also defines a set of function pointers called at key points during GCC's processing of a compilation unit, these function pointers can be set in `gem_create`. The function pointers of particular interest to us are: `gem_start_function`, `gem_finish_function`, `gem_finish_decl`. These function pointers are set in the `gem_create` function of our extension module and allow us to access the syntax trees for functions and global declarations. It should be noted that GCC does not keep an explicit symbol table in its high level intermediate forms, rather properties normally associated with the symbol table are stored as attributes of syntax tree nodes. For example, given a variable declaration node of a syntax tree, its identifier can be accessed using the `DECL_NAME` macro which simply accesses the appropriate field containing the address of the identifier node.

At the point where the syntax trees are intercepted by GEM they are still in a high level form. Most – if not all – of the original constructs of the language – in this case C – are still intact such as compound literal expressions, all of the various looping constructs, etc.. This form is rather tedious to deal with as distinguishing between for and while loops or dealing with nested assignments is not productive for our purposes. In fact, dealing with this form directly would result in excessive code duplication. Luckily, as of GCC 4, there is an intermediate form called GIMPLE form which is like a high level three address form. By copying the intercepted syntax trees and converting them to GIMPLE form we can work with a simplified syntax tree that still preserves a great deal of semantic information.

Note that despite the simplicity of GIMPLE form it still maintains semantic information such as the types for variables. Once in GIMPLE form, the assignments and function calls are picked out and written to a compilation unit summary. As a small optimization, assignments not involving a pointer type or some kind of aggregate type (such as a struture) are not recorded. Since flow control statements and the like are not relevant to the FA analysis they are simply ignored. In the next section, greater detail will be given to the specifics of the compilation unit summaries.

5.2 Compilation Unit Summaries

Our software uses an XML markup for producing the compilation unit summaries. The document contains three sections: a global assignment section, a section for each function, and a type table section. The main purpose of these sections is to categorize both where assignments happen and, with the use of the type information, what aliasing might occur when aggregate types are involved. Before going into extensive detail of each section, we start by looking at the representation of an object name.

The construction of object names in the FA analysis are mapped to an equivalent XML format. A variable/heap name is represented by a `<decl>` tag, a field access by a `<field_access>` tag, a dereference by an `<indirect>` tag, and an address operator by a `<address>` tag. To distinguish between two variables of the same name but located in different scopes – such as being in different functions, or fields of different structure types, etc. – we introduce a `<namespace>` tag which contains the object name of the scope to which a variable belongs. So, for a variable local to a function, its `<namespace>` tag would contain the object name for the function to which it belongs. If the variable in question is actually the field of some structure type then the `<namespace>` tag will contain either a `<record>` or `<union_type>` tag to denote the type of structure to which it belongs. Global variables simply contain an empty `<namespace>` tag or, if the variable is file scope, the name space tag contains a `<file>` indicating the file to which it has scope.

To make this more concrete. Consider the object name `(*next).prev` where next is a local variable of the static function `__list_add` in `file_table.c` and `prev` is a field of structure type `list_head`. This object name would be represented in our summaries as in Figure 5.

```
<field_access>
  <indirect_ref>
    <decl id="next">
      <namespace>
        <decl id="__list_add">
          <namespace>
            <file id="file_table.c">
              <namespace/>
            </file>
          </namespace>
          <type-ref key="770545756"/>
        </decl>
      </namespace>
      <type-ref key="770545687"/>
    </decl>
    <type-ref key="770545664"/>
  </indirect_ref>
  <decl id="prev">
    <namespace>
      <record_type id="list_head">
        <namespace/>
      </record_type>
    </namespace>
    <type-ref key="770545687"/>
  </decl>
  <type-ref key="770545687"/>
</field_access>
```

Figure 5: XML representation of an object name.

The FA analysis expects assignments to be in the form of $A = B$ where A

and B are both object names as outlined in the FA analysis. GIMPLE form presents assignments in the form $a = b\ op\ c$ where *op* is some valid C operator. Since GIMPLE form ignores pointer arithmetic, this generally transforms easily into the representation expected by the FA analysis. One issue that does arise is the initialization of global variables. Since GCC provides no facility to transform the initialization expressions of global variables into GIMPLE form, some high level constructs must be taken in to account. This involves taking a high level C construct and breaking it down into a simple series of assignments. For example, a constructor assignment such as Figure 6 is transformed into a series of assignments shown in Figure 7.

```
static struct inode_operations umsdos_file_inode_operations = {
   truncate:    fat_truncate,
   setattr:     UMSDOS_notify_change,
};
```

Figure 6: Constructor assignment code.

```
umsdos_file_inode_operations.truncate = fat_truncate;
umsdos_file_inode_operations.setattr = UMSDOS_notify_change;
```

Figure 7: Constructor assignment broken into multiple simple assignments.

Assignments are represented in the summaries as simply an `<assignment>` tag whose first child is the object name for the left had side of the assignment and whose second child is the the object name for the right hand side of the assignment. The right hand side may also be a function call as described next.

Function calls in the source program are represented in the summary using a `<function_call>` tag. The function call is appropriately transformed – likely through the introduction of temporary variables – such that each parameter passed is a single object name and not an assignment or function call. The first child of the `<function_call>` tag is the object name of the callee; the callee may either be an address of a function object name (for direct calls) or a pointer object name (for indirect calls). All subsequent children are `<param>` elements, each with a number attribute indicating the argument position. Each `<param>` element consists of a single object name representing a parameter of the function call. For example, compare the function call to `open_namei` in Figure 8 to its XML representation in Figure 9.

For each function in the compilation unit there is a summary for that particular function denoted by a `<function>` tag. The first child of a `<function>` tag is the object name of that function. For a function with n parameters the 2nd through (n+1)th child of the `<function>` tag are object names for the functions parameters in order. Next is the object name of the function's return value. The rest of the `<function>` tags children are either `<assignment>` tags or

```
int open_namei(const char *pathname,
               int flag,
               int mode,
               struct nameidata *nd);

struct file *flip_open(const char *filename, int flags, int mode){
int namei_flags, error;
struct nameidata nd;

// some code

struct nameidata *t01 = &nd; // temp inserted by GIMPLE
int error = open_namei(filename, namei_flags, mode, t01);

// some more code
}
```

Figure 8: Code snippet from the Linux file system code.

`<function_call>` tags. The representations of assignments and function calls in the function summary are presented in the same order as they appear in the source program.

All the types in the compilation unit are stored in the type table section, denoted by a `<type-table>` tag, at the end of the compilation unit summary. The type table itself is just a large hash table mapping integer keys to type values. Every object name contains a `<type-ref>` tag with a key attribute that corresponds to its type in the type table. Each value in the type table may contain references to other values in the type-table and may even have circular references for recursive types. The children of the `<type-table>` tag are `<value>` tags each with a key attribute. The `<value>` tag contains exactly one child which is one of several type tags. Function types are denoted with a `<function-type>` tag, structure types with a `<record-type>` tag, union types with a `<union-type>` tag, integer type with an `<integer>` tag, real type with a `<real-tag>` and an array type with an `<array-type>` tag.

A `<function-type>` tag's first n children correspond to types of parameters for a function of that type and the last child corresponds to the type of the return value type of a function of that type. The `<record-type>` tag and `<union-type>` tag have a sequence of object names for their fields in the source program. These fields appear in exactly the same order as in the source program. The `<integer>` type tag has no children but does have a size and signed attribute. These attributes correspond to the size of the integer type in bits and a boolean value of which is true if the integer type is signed and false otherwise. The `<array-type>` has exactly one child corresponding to the type of each element of the array.

```
<assignment>
  <decl id="error">
    <namespace>
      <decl id="flip_open">
        <namespace />
      </decl>
    </namespace>
  </decl>
  <function_call>
    <address>
      <decl id="open_namei">
        <namespace />
      </decl>
    </address>
    <param num="0">
      <decl id="filename">
        <namespace>
          <decl id="flip_open">
            <namespace />
          </decl>
        </namespace>
      </decl>
    </param>
    <param num="1">
      <decl id="mode">
        <namespace>
          <decl id="flip_open">
            <namespace />
          </decl>
        </namespace>
      </decl>
    </param>
    <param num="2">
      <decl id="mode">
        <namespace>
          <decl id="flip_open">
            <namespace />
          </decl>
        </namespace>
      </decl>
    </param>
    <param num="3">
      <decl id="t01">
        <namespace>
          <decl id="flip_open">
            <namespace />
          </decl>
        </namespace>
      </decl>
    </param>
  </function_call>
</assignment>
```

Figure 9: XML representation of Linux file system code snippet.

Figure 11 shows the XML representation of a type table consisting of types depicted in Figure 10. These types are loosely based off types from the Linux kernel's filesystem code.

```
struct file{
   struct dentry *f_dentry;
   struct file_operations *f_op;
};

struct file_operations {
   int (*open)();
};

struct dentry {
   struct dentry *d_parent;
};
```

Figure 10: Partial structure definitions from Linux file system code.

5.3 FA Analysis Implementation

Since our compilation unit summaries essentially insulate the implementation of our analysis from the software used to gather information about the C code, we are not locked into using any specific language in writing the FA analysis. Do to time restraints, the implementation was written as a Python script. This allowed for the analysis itself to be written relatively quickly and made debugging an easier affair than if it had been written in C/C++. This has, however, come at a cost as it currently takes upwards 1 hour to process the entire 2.4 Linux kernel even with a minimalist kernel configuration. The inefficiencies of Python forced us – somewhat ironically – to be more conscientious about efficiency during implementation.

The approach used is loosely object oriented. Every assignment, object name, function call, type, etc. maps to an object of an appropriate class. Figure 12 presents the class diagram for object names. Types have a similar mapping.

Since both object names and types are immutable and multiple occurrences of the same object name/type occur quite frequently, it becomes quite costly in terms of memory to create a new object for each object name. To deal with this issue, object construction for classes related to object names and types has been customized so that a cache is checked prior to object creation. If the cache already contains an appropriate instance for the object name or type, this instance is returned and no object creation occurs. This insures that there are never multiple instances of equivalent object names or types. This not only significantly reduces the memory footprint but it also significantly reduces the run time as equality checks can be done based on memory addresses using

```
<type-table>
  <value key="1">
    <record-type>
      <decl id="dentry">
        <type-ref key="5"/>
      </decl>
      <decl id="f_op">
        <type-ref key=""/>
      </decl>
    </record-type>
  </value>
  <value key="2">
    <record-type>
      <decl id="open">
        <type-ref key="7"/>
      </decl>
    </record-type>
  </value>
  <value key="3">
    <record-type>
      <decl id="d_parent">
        <type-ref="5"/>
      </decl>
    </record-type>
  </value>
  <value key="4">
    <integer-type size="32" signed="1"/>
  </value>
  <value key="5">
    <pointer-type>
      <type-ref key="3"/>
    </pointer-type>
  </value>
  <value key="6">
    <pointer-type>
      <type-ref key="2"/>
    </pointer-type>
  </value>
  <value key="7">
    <function-type>
       <type-ref key="4"/>
 <void-type>
    <function-type>
  </value>
  <value key="8">
    <pointer-type>
      <type-ref key="4"/>
    </pointer-type>
  </value>
</type-table>
```

Figure 11: Partial XML representation for Linux file system types.

Python's "is" operator, as opposed to checking equality by recursing on object names/types.

Caching object name instances is simple enough, since an object name is either a variable/heap name, an object name with an address "&" or indirect "*" prefix, or a field access "." of two object names. For each class that inherits from ObjectName, such as FieldAccess, Address, etc. we store a hash table and in this hash table we store instances of that class using the hashes of their children as the key. For example, Figure 13 shows the constructor for the FieldAccess class which performs this caching.

Caching type instances can be more difficult as many types are recursive, such as the structures often used for linked lists. This circular dependency means we cannot simply use the subtypes of a given type as a key to a types instance. Consequently, when a compilation unit summary is parsed, we allow for multiple instances of the same type to exist, but once a compilation unit is parsed, we prune out these duplicates.

Originally, the XML summaries were parsed in as a DOM using Pythons

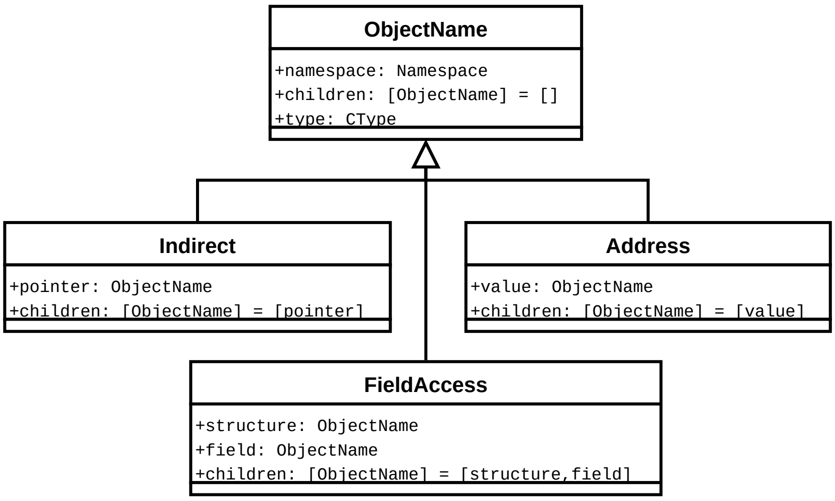

Figure 12: UML class diagram for object names.

XML reader and the DOM objects were mapped into our internal representation for object names, assignments, function calls, etc. While this was sufficiently fast for small programs, using this method to process all of the summaries for the Linux Kernel took upwards six hours. Consequently, the parsing was rewritten to use an XML SAX parser. SAX does not build any tree structures corresponding to the parsed XML document, it simply issues events whenever a tag or attribute is encountered. We use these events to build up our own internal representation for the summaries which is fairly trivial given the near isomorphism between the internal representation we use and the structure of the XML summaries. The code becomes slightly more complex and prone to error due to pushing and popping objects from a stack, but by using this method, it now only takes approximately one hour to parse the entire Linux kernel as opposed to six.

The algorithm itself is implemented as closely as possible to the pseudo code provided by [12]. The FA analysis uses a PE equivalence relation (Pointer-related Equality) to categorize all object names into equivalence classes. If two object names are in the same equivalence class then they may alias one another. If two object names are not in the same equivalence class then they will not alias each other. With each equivalence class there is an associated prefix which is essentially a labeled edge list for pointer dereferences and field accesses. Using the classes PEGraph and PENode (Figure 14) we represent these equivalence classes and their corresponding prefixes as an explicit graph.

The methods `init_equiv_class`, `find` and `merge` correspond to the functions init_equiv_class, find and union presented for the FA Analysis. These methods behave in the expected manner given the functions in the FA analysis they represent, so we will not go into extensive detail of them here. However,

```
class FieldAccess(ObjectName):
    instance_dict = dict()

    # objname is an instance of ObjectName occurring on
    # the left hand side of a field access.
    # field is an instance of ObjectName, or more
    # specifically Declaration, and occurs on the right
    # hand side of the field access.
    def __new__(cls, objname, field):
        try:
            # If we have already constructed a FieldAcess
            # for objname and field go ahead and return it.
            return FieldAccess.instance_dict[(objname,field)]
        except KeyError:
            # We have not yet constructed an instance of
            # FieldAccess for objname and field so lets
            # make one now.
            instance = ObjectName.__new__(cls)
            ObjectName.__init__(instance)
            # Here we store this instance for reuse.
            FieldAccess.instance_dict[(objname,field)] = instance
            instance.children = [objname,field]
            instance.namespace = objname.namespace
            instance.__calc_hash_value()
            return instance
```

Figure 13: Modified constructor to cache instances.

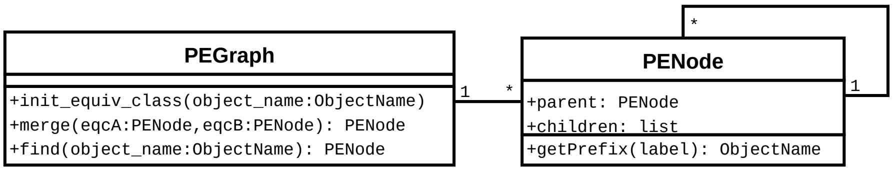
PEGraph
+init_equiv_class(object_name:ObjectName)
+merge(eqcA:PENode,eqcB:PENode): PENode
+find(object_name:ObjectName): PENode
1
*
PENode
+parent: PENode
+children: list
+getPrefix(label): ObjectName
*
1

Figure 14: UML class diagram for PENode and PEGraph.

there are some important considerations to make when using these methods. All object names (an `ObjectName` instance) should be initialized prior to their use by find otherwise an exception is raised. Also, the return value of `find` is a `PENode` representing the equivalence class to which it belongs. This value is only valid until the next call merge or `init_equiv_class` as the object name's equivalence class (and thus `PENode`) may change as a side effect of these method calls. The union find algorithm is utilized to handle both lookups and merges of equivalence classes.

`PENode` provides a method getPrefix which, given a field `ObjectName` instance or the `INDIRECT` constant, will return an `ObjectName` instance for the prefix of that field access or indirection respectively. This corresponds to the apply function of the FA analysis.

The implementation of the FA algorithm itself is fairly straight forward from the algorithm's pseudo-code. There are some additions to the algorithm in-order to better serve our particular needs. We reproduce an outline of the algorithm here in Figure 15.

Notice that in our implementation of the FA analysis we do the initialization of object names and the merging of regular assignments while we are still parsing compilation unit summaries. This simply reduces the total amount of time needed to run the analysis on large pieces of software such as the Linux kernel, which, even in our minimalist configuration, contains nearly 300 compilation unit summaries.

The function `propagate_inlined_functions` simply accounts for function inlining by associating the the function calls contained in an inlined function with the callees of the inlined function. The purpose of removing inlined function at this point is to better serve the needs of machine code level optimizations which will have no knowledge of inlined functions. The burden of dealing with function inlining could be placed upon the machine level analysis but this would require the use of DWARF debugging information.

The function `write_callgraph` simply dumps the callgraph to the given file handle. The callgraph written is only a partial callgraph, only including edges for the indirect calls. It is necessary for any analysis using these results to fill in the gaps for normal function calls.

5.4 External and Inline Assembly

As discussed earlier, high level alias analyses depend on the program being written entirely in a high level language. The Linux kernel, however, contains a great deal of inline assembly and external assembly files. To solve this problem, we make use of approximate decompilation which was developed by Haifeng He at the University of Arizona. We will briefly sketch his approach here, for a more detailed discussion see [9].

Approximate decompilation transforms assembly code in to C code but does not do so to preserve the exact computation. Instead, approximate decompilation only attempts to preserve particular properties of the assembly code during the translation. In our case, we wish to use the FA analysis on the translated assembly code of the kernel. So the properties that must be maintained during the approximate decompilation are those properties upon which the FA analysis depends.

The FA analysis is flow-insensitive and context-insensitive so intra-procedural control flow instruction in the assembly can be ignored. This includes branch instructions that do not transfer control outside of a given function. However, any instruction which transfers control to another function – indicative of a function call – must be preserved as the FA analysis needs to merge the parameters of the call site with the formal parameters of the callee.

Instructions that move data around – all of which could possibly be memory addresses – are translated into assignments. System instructions can simply be ignored since they only manipulate hardware which does not affect alias analysis.

5.5 Running the Analysis

To produce a partial call graph for the Linux kernel requires four steps:

1. Add GEM Patched GCC to the Path
2. Modify kernel Makefile
3. Compile kernel
4. Run FA Analysis Python Script

Since we use a version of GCC with the GEM patch applied for loading extension modules, we have to put this version of GCC on the path. Placing it before other versions of GCC is sufficient to insure that the patched version of GCC is used. We will assume that this version of GCC is in `/home/trimblej/gcc-gem` but it could of course be else where. In the bash shell we would set the path as follows:

```
$ export PATH=/home/trimblej/gcc-gem/bin:$PATH
```

Modification of the kernel's Makefile is also necessary in order to add the appropriate command line argument to load our function summary GCC extension module. To make absolutely certain that the extension module is loaded, we modify the `CC` variable as opposed to one the `CFLAGS` variables. Assuming that the extension module is located at `/home/trimblej/module.gem`, we change the following line in the root Makefile of the kernel from:

```
CC = $(CROSS_COMPILE)gcc
```

To:

```
CC = $(CROSS_COMPILE)gcc -fextension-module=/home/trimblej/module.gem
```

The kernel is then compiled in the usual way using a configuration file that includes only the bare essentials needed by the hardware and software applications being used. After the kernel is compiled, the source tree will contain a large number of files with an `.xml` extension. These files are the compilation unit summaries. These summaries need to be passed as command line arguments to the `fa_analysis.py` script which will run the FA analysis on them. The simplest way to do this is to go to the kernel's root directory and use the *find* utility to grab all the `.xml` in the kernel's source tree. Assuming that the script is located at `/home/trimblej/fa_analysis.py` and the kernel's source at `/home/trimblej/kernel`, this would be executed as follows from the kernel's root directory:

This will write to a file called `callgraph` in the current directory which will contain for each function with indirect calls its potential targets as a result of those indirect calls. This is then used the binary rewriting code compaction software to compact the kernel.

6 Experimental Results

Figure 16 displays the number of possible call targets for each indirect call. There are 1092 indirect calls in total in the compiled Linux kernel we configured.

For code compaction of the kernel, a variety of benchmarks are taken from the MiBench suite [8]. These benchmarks simulate the environment of an embedded system. A brief summary of the programs used in each benchmark is given in Table 1. The compaction results of for each benchmark is given in Table 2 which includes the results both with and without the FA analysis data.

```
$ python ../fa_analysis.py ‘find -name "*.xml"‘
```

Benchmark	Programs
Boot	
Automotive	*basicmath, bitcount, qsort, susan*
Consumer	*jpeg, lame, mad, tiff2bw, tiff2rgba, tiffdither, tiffmedian, typeset*
Network	*dijkstra, patricia, CRC32, sha, blowfish*
Office	*ghostscript, ispell, rsynth, sphinx, stringsearch*
Security	*blowfish, pgp sign, pgp verify, rijndael, sha*
Telecomm	*CRC32, FFT, IFFT, ADPCM, GSM*
Cellphone	*blowfish, sha, CRC32, FFT, gsm, typeset*
Entertainment	*jpeg, lame, mad*

Table 1: Test benchmarks.

Benchmark	With FA Analysis	Without FA Analysis	Difference
Boot	72.5	77.2	4.7
Automotive	72.8	77.7	4.9
Consumer	73.1	78.1	5.0
Network	72.8	77.8	5.0
Office	73.0	77.9	4.9
Security	72.9	77.8	4.9
Telecomm	72.8	77.8	5.0
Cellphone	72.9	77.9	5.0
Entertainment	73.0	78.0	5.0

Table 2: Linux 2.4 kernel code compaction results as percentage of uncompacted code.

7 Experimental Evaluation

Table 16 shows some rather interesting results. First of all, out of the 1092 indirect calls found 752 of them – over half – were found to have exactly 1 target. So the accuracy of most indirect calls is a drastic improvement over using a low level alias analysis. However, 82 indirect calls have over 100 possible targets and, out of those, 20 have over 200 possible targets.

All of the indirect calls containing over 200 possible targets are due to the same set of function targets. This set includes all of the system call handlers such as as `sys_clone`, `sys_settimeofday`, `sys_mount`. Some others in this group appear to be be error handlers such as `coprocessor_segment_overrun` and `divide_error`. The rest appear to deal with the file system such as `ext2_read_inode` and `proc_read_inode`. Two of the indirect calls in the this set are not particularly surprising, one is the function `system_call` and the other `error_code`. The other indirect calls are in functions related to the filesystem such as `dcache_readdir`, `proc_readfd`, and `ext2_readdir`. It is curious to see specific filesystem functions like `ext2_read_inode` lumped together with system call handlers though this may be accounted for by how Linux treats everything as a file.

Those indirect calls having over 100 targets but less than 200 are also all due to a particular set of functions. These all appear to deal with I/O either in the filesystem code, such as `ext2_mkdir`, to various peripherals, such as `busmouse_read`, or to memory, such as `memory_open`. Most of the indirect calls themselves are spread throughout the filesystem code such as in `dentry_open` and `vfs_symlink`. Some of the calls are in system call handlers such as `sys_pread` and `sys_read`. We can rationalize the size of this call target set in a similar way as the previous such set: since almost everything in Linux is represented as file it should not be overly suprising that filesystem code and more general I/O code would get lumped together.

Establishing exactly how precise the FA analysis is in determing the potential targets of indirect calls in the Linux kernel is something of a problem. Milanova et al. established the accuracy of the FA analysis in finding potential targets for indirect calls by using a small set of applications for which they, by hand (!), constructed the callgraph. They then compared the results of the FA analysis to this ideal, hand made, call graph [12]. Clearly this is impractical for the Linux kernel. To give some notion of the precision we can compare to the fairly primitive aliasing information gained when performing code compaction without the FA analysis. With out the FA analysis, every indirect call has 1001 potential targets, whereas the FA analysis produces an average of 14 potential targets for each indirect call.

As far as the FA analysis' usefullness in code compaction, all of the benchmarks show an average compaction improvement of about 5% (Table 2) over performing code compaction without the FA analysis.

8 Related Work

Software written in object-oriented languages such as Java and C++ tend to have a great deal of indirect calls as a result of virtual methods. Consequently, a great deal of research has evolved concerning callgraph construction in object-oriented languages [13, 7, 2, 1]. The techniques used to resolve indirect call targets in object-oriented languages tend to be largely type based. Initially we attempted to apply a type based approach to resolving indirect calls in the kernel but the initial results were not satisfactory. The main barrier to applying such type based approaches to the kernel code is that C is not a strongly typed language. For example, in C, a pointer of type *a* can be implicitly cast to a pointer of type *b* with only a warning from GCC by default.

Resolving indirect call targets is an essential task for code compaction. Existing work does alias analysis at the same level as the code compaction. Debray et. al is the earliest instance found of low level code compaction using control flow analysis [5]. Their work uses binary rewriting for compaction and resolves indirect calls conservatively by assuming that an indirect Call's target is potentially any function whose address is taken. Chanet et al. uses a similar binary rewriting technique for code compaction of the Linux kernel and uses essentially the same method for indirect call resolution [3].

9 Conclusion

Combining high level aliasing information with low level code compaction of the Linux kernel is both feasible and benificial. Combining source level aliasing information with low level code compaction of the Linux kernel can be achieved by having the source level analysis build part of the kernel call graph. The kernel's source code can be processed by using a GCC plugin and relevant information stored as compilation unit summaries which are written in XML. These summaries can then be used to run the FA analysis. Even though the FA analysis is one of the least precise alias analysis algorithms its results are able to improve code compaction of the kernel by 5%.

10 Future Work

Currently the granularity of the aliasing information from the source level provided to low level code compaction is only at the function level. This in turn limits the level at which unreachable code can be eliminated. By providing more general aliasing information about specific memory references finer grained code compaction could be accomplished. The main barrier to this is developing a mechanism by which the source level aliasing information can be effectively communicated to low level code compaction. Using DWARF debugging information may provide one avenue by which such can be accomplished.

11 Acknowledgments

I would like to thank my advisor Dr. Debray and Dr. Andrews for all of their help and encouragement in my undergraduate research here at the University of Arizona. I would also like to thank Haifeng He and Somu Perianayagam for all their help and patience in tracking down bugs in my code.

References

[1] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. *Lecture Notes in Computer Science*, 1098:142–??, 1996.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. pages 324–341.

[3] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide compaction and specialization of the linux kernel. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 95–104, New York, NY, USA, 2005. ACM Press.

[4] Tzi cker Chiueh and Alexey Smirnov. Gem: Gcc extension modules, 2006.

[5] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[6] Saumya K. Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Symposium on Principles of Programming Languages*, pages 12–24, 1998.

[7] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124, New York, NY, USA, 1997. ACM Press.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. pages 3–14, December 2001.

[9] Haifeng He, John Trimble, Somu Perianayagam, Saumya Debray, and Gregory Andrews. Code compaction of an operating system kernel. In *Symposium on Code Generation and Optimization*, 2007 (to appear).

[10] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[11] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[12] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise and efficient call graph construction for c programs with function pointers.

[13] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM Press.

[14] S. Zhang. *Practical Pointer Aliasing Analyses for C.* PhD thesis, 1998.

[15] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Foundations of Software Engineering*, pages 81–92, 1996.

```
def run_fa_analysis(graph, files):
  num_files = len(files)
  functions = []
  function_calls = []
  # Do file parsing in a separate thread
  parser_thread = FAParserThread(files)
  parser_thread.start()

  while num_files > 0:
    num_files -= 1
    # Get a compilation unit summary.
    summary = parser_thread.results.get(True)
    functions.extend(summary.functions)
    for function in summary.functions:
      # initialize function, argument, and return
      # value object names.

                ⋮

    for assignment in summary.assignments:
      # initialize the object names of the left
      # hand and right hand side of the assignment.
      # merge left hand side and right hand side
      # object names.

                ⋮

    function_calls.extend(summary.function_calls)
    for function_call in summary.function_calls:
      # initialize target object name.
      # initialize argument object names.
      # initialize assignee object name [if return value saved].

                ⋮

  for function_call in function_calls:
    # Resolve function call targets and merge arguments
    # with functions formals.

                ⋮

  # Handle inlining of functions.
  propagate_inlined_functions(functions, function_calls)
  # Dump the callgraph.
  f = open("callgraph", "w")
  write_callgraph(functions, function_calls, f)
  f.close()
```

Figure 15: FA algorithm implementation outline.

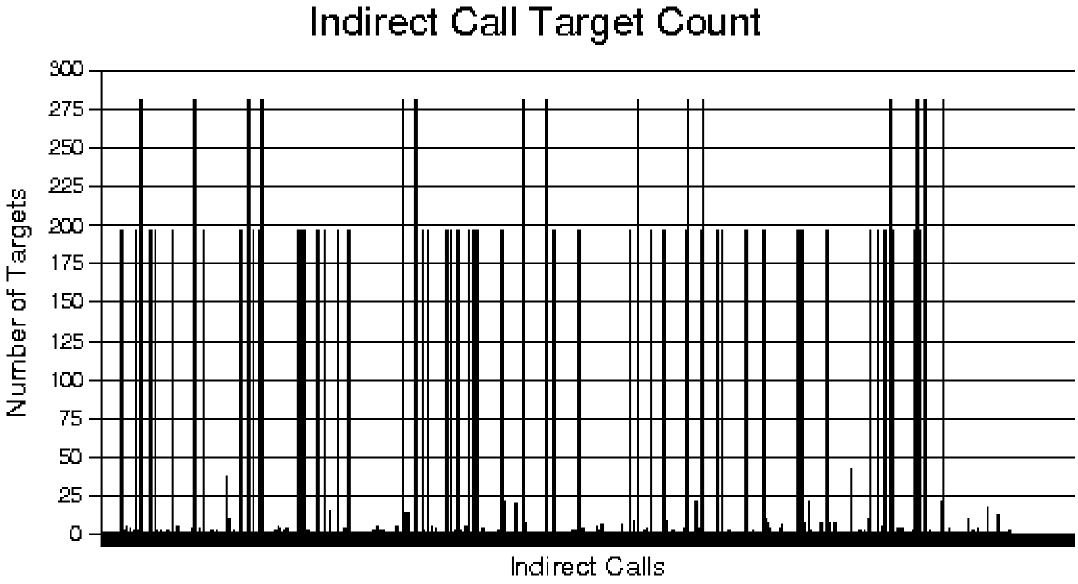

Figure 16: Indirect Call Target Count.