Automatic Data Environment Construction for Static Device Drivers Analysis

(Extended Abstract)

Hendrik Post, Wolfgang Küchlin
University of Tübingen / Symbolic Computation Group
72076 Tübingen, Germany
{post,kuechlin}@informatik.uni-tuebingen.de

ABSTRACT

Linux contains thousands of device drivers that are developed independently by many developers. Though each individual driver source code is relatively small—≈10k lines of code—the whole operating system contains a few million lines of code. Therefore Linux device drivers offer a useful application area for modular analysis.

Our finding is that despite the precise modeling of most features of the standard systems programming language C, model checking software verification tools for C fail to provide means for modular analysis of device drivers. We inspected CBMC [2], SLAM-SDV [3], MAGIC [1], BLAST [4] and others and found that a rich additional environment model for every device driver is needed. This model must provide information on out-of-scope initialized pointers and complex data structures. We present strategies to automatically create feasible, bounded data environments for Linux device drivers instead of creating them manually. Our solution differs from general interface generation mechanisms (e.g. CUTE[5]), because is it specialised on bounded model checking of Linux device drivers written in C. Our contribution is a preprocessing step that extends the usability of CBMC for modular Linux device driver analysis.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Verification—*Model Checking*; D.4.5 [**Operating Systems**]: Reliability—*Verification*

General Terms

Verification, Experimentation

Keywords

Linux, Bounded Model Checking, Environment Modelling, Software Verification

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

1. INTRODUCTION

Software verification is commonly interpreted as the analysis of a software system concerning latent or possible errors. Though the degree of precision and the tradeoff between completeness and a low false-positive rate differs, a common pragmatic aim of verification is to find and eliminate errors. As verification requires lots of resources, it is preferably applied to systems where a stable, specified behavior of the software is of high importance.

The problem we identify with modular verification of Linux device drivers is external data representation. Data initialization and data usage are performed on multiple operating system layers. The data interface between layers includes pointers, and especially those that are subject to pointer arithmetic. Hence modular analysis faces the problem that pointers are used extensively, but exact information about their initialization or targets is commonly not available within the scope of analysis as discussed in Section 2.2.

As many tools for the analysis or even verification of C programs exist, we aim to provide preprocessing such that one of these can be used on device drivers. Our suggestion is to provide concrete targets for all externally initialized pointers, whereas all external variables of primitive data types remain unconstrained symbolic values. This approach is therefore located between pure abstract static analysis and concrete software testing.

The tools for the analysis of C programs we explicitly reviewed are CBMC[2], BLAST [4], Meta-Compilation [9], SATURN [8], MAGIC[1] and the Static Driver Verifier from the SLAM project [3]. The modular analysis tool MAGIC for example does not support dereferencing of pointers on the left side of an assignment at all. The analysis performed with BLAST in [4] has been made possible by writing a test driver. The static driver verifier from the SLAM project [3] also requires a manually created operating system model prior to analysis. The Meta-Compilation project also put an extensive amount of work into the generation of a test environment via abstracting the Linux kernel [9].

Several sophisticated tools for environment generation exist, though they target different or more general application areas and hence provide different strategies. Symstra[7] and CUTE[5] seem closest to our approach. Symstra generates environments for static analysis as we do, but its current implementation covers only programs written in Java. CUTE includes a mechanism to expand the object graph on the fly, should its analysis indicate the necessity. Its approach is a mixture of Symbolic Execution and Testing while we aim

to provide only preprocessing for model checking tools. As both tools target general applications instead of the limited area of Linux device drivers, a direct support of common Linux abstract data types was not considered. Instead of a general approach we present a minimal environment generation in order to facilitate the analysis of device drivers.

We chose the software verification tool CBMC as our analysis backend due to its extensive support of C language features as summarized in Section 2.1. We will also discuss problems when using CBMC for modular analysis.

2. VERIFICATION TOOLS AND TARGETS

2.1 CBMC

CBMC 2.1 is a bounded model checker intended to be used for the analysis of C programs and Verilog descriptions. When running in C analysis mode, it translates ANSI-C programs into propositional logic. Loops and recursion are handled by code unwinding. CBMC supports pointer arithmetic, integer operators, type casts, side effects, function calls, calls through function pointers, non-determinism, assumptions, assertions, arrays, structs, named unions, and dynamic memory. Therefore this tool is a good choice to analyze systems code written in C that makes use of these features. CBMC itself is capable of finding double-free and free-after-use errors beside bounds and pointer validity checking.

CBMC offers an extensive treatment of pointers that essentially tracks the object and the offset a pointer points to. Nevertheless two technical problems remain unsolved in the documentation. First, pointers—when dereferenced—must point to a valid object. Though this is a reasonable assumption for the runtime behavior of programs, it is not useful for modular static analysis of programs where targets of pointers are often unknown. The second shortcoming is the modeling of possible aliases between pointers to unknown targets. We illustrate both problems by a short example:

```
 1: struct person_t {
 2:   int age;
 3: } a_person;
 4: void set_age_difference(struct person_t* p1,
          struct person_t* p2, int diff) {
 5:    p2->age = p1->age + diff;
 6:    assert(p2->age == p1->age + diff);
 7: }
 8: void main() {
 9:    struct person_t* p1 = &a_person;
10:   struct person_t* p2 = &a_person;
11:   set_age_difference(p1, p2, 20);
12: }
```

The example is artificial for the sake of simplicity. In function `set_age_difference()` two structs are passed by reference. The age of the second person's record should be set to the age of the first person's record plus an age difference. Line 6 contains a check if the assignment was made correctly. The assertion is invalid because both pointers may alias each other. In this case both dereferences of the age fields are equal which violates the assertion if `diff` $\neq 0$. The two different entry points for an analysis are `main` and `set_age_difference`. Moreover CBMC allows to either enable or disable checks for invalid dereferences of pointers. The latter feature is almost undocumented. We assume that disabling pointer checks globally disables checks for `null` or uninitialized pointers. We identify four different analyses entering the module either at `main` or `set_age_difference` and either checking for invalid pointers.

1. *Entry at* ***`main`****, Pointer Checks disabled.* When CBMC starts at `main`, `p1` and `p2` are explicitely aliased and passed as a reference to `set_age_difference`. Hence dereferencing them is allowed and writing to one of the pointer targets correctly modifies the aliased pointer dereferences as well. The assertion is invalid as expected.

2. *Entry at* ***`main`****, Pointer Checks enabled.* This case leads to the same result as case 1.

3. *Entry at* ***`set_age_difference`****, Pointer Checks enabled.* This direct entry reflects the interleaving of modules within the Linux kernel where dispatch routines may be directly called from outside the module, and the environment as provided by `main` is unknown. When `set_age_difference` is analyzed with enabled pointer checks, CBMC correctly emits a warning that dereferencing the parameters in line 5 might be incorrect. This is due to the correct modeling that unconstrained pointers may be `null`. Though the result is technically correct it seems more reasonable for modular analysis to assume that the environment is correctly initialized.

4. *Entry at* ***`set_age_difference`****, Pointer Checks disabled.* Dereferencing the parameters is not a problem any more, but nevertheless the assertion is fulfilled. This is an incorrect analysis result when the scope of analysis is limited to `set_age_difference`. CBMC does not model a possible aliasing between uninitialized pointers.

Cases 3 and 4 offer significant shortcomings to device driver analysis. The impact on our work is twofold.

The user of CBMC has two choices which both lead to disadvantages: the user may globally disable pointer checks which might only be desirable for interface pointers, but not for pointers manipulated inside the module. Second, the user of CBMC faces the problem that alias relationships induced by an unknown operating system environment are not modelled or taken into account. This leads to an even less appropriate analysis. We assume that interface objects are set up correctly and hence we must perform the correct initialization before calling the entry function of the module.

2.2 Linux Device Driver Interfaces

Linux device drivers often operate on structs, each of which represents one device [6]. The generic CD-Rom device driver `cdrom.c`, for example, may service several hardware drives, each represented by one `struct cdrom_device_info`. This struct is of course passed by reference to all service routines found in the driver. The struct is partially listed in Figure 1. Devices are organized in a linked list via the field `struct cdrom_device_info * next`. When another system layer steps into the driver dispatch routines, it passes the currently serviced device via a reference to a struct. The struct itself is not initialized within any cdrom driver, but in other system layers. Invoking a verification tool on any service routine in `cdrom.c` leads to a problem. It is not evident

```
struct cdrom_device_ops {
   int (*open) (struct cdrom_device_info *, int);
   void (*release) (struct cdrom_device_info *);
   int (*drive_status)
               (struct cdrom_device_info *, int);
   ...
};
...
/* Uniform cdrom data structures for cdrom.c */
struct cdrom_device_info {
   struct cdrom_device_ops  *ops;
   struct cdrom_device_info *next;
   struct gendisk *disk;
   ...
   int speed; ...
};
```

Figure 1: The operation interface struct and the `cdrom_drive_info` struct listed from file `drivers/cdrom/cdrom.c` of Linux kernel 2.6.15.

that all references passed as a parameter are correctly initialized. Hence CBMC would correctly emit a warning that dereferencing one of them may lead to a potential `null` dereference. Though this message may be suppressed the better assumption would be to assume that the device structs are properly initialized.

In the next section we give a solution to this problem using automatically created data environments.

3. CREATING DATA ENVIRONMENTS

The general approach to create a suitable environment is to identify potentially uninitialized pointers that are either declared in the global scope of the module, or parameters to the analyzed entry point in the module. For each pointer, a fresh object can be created and the pointer is initialized by pointing to this object. This strategy is also used in Symstra[7] and CUTE[5]. This could lead to new uninitialized pointers if the object created is of a pointer type, contains fields of pointer type or is an array with elements of pointer type. In all three cases fresh objects are created for these pointers as well, up to a bounded object graph depth. This depth-bound reflects the size of a recursive data structure, e.g. a list. For binary trees this bound limits the depth of the tree. For bounded model checking this depth should be smaller than or equal to the CBMC bound for loops and recursion, as a loop iterating over all elements of a list or a recursive search for an element in a tree won't violate the unwinding assertions. Then CBMC may capture the exact behavior of functions on bounded data structures.

We identify two sources of pointers that must be initialized:

1. Pointers within the global scope of the translation unit.
2. Parameters of pointer type within the module entry function to be analyzed.

The main algorithm performs a breadth-first search on the object graph with bounded depth:

```
 1: worklist = union(global_pointers(file),
       parameters(entry_function));
 2: object stub_obj; depth = 1;
 3: new_worklist = {};
 4: while (!is_empty(worklist)
                  && depth < depth_bound) {
 5:   for each pointer p in worklist {
 6:       stub_obj = create_object_for_pointer(p);
 7:       create_assignment_for(p,stub_obj);
 8:       new_worklist = union(new_worklist,
                        pointer_members(stub_obj);
 9:   }
10:  worklist = new_worklist;
11:  new_worklist = {};
12:  depth = depth +1;
13:}
```

The search for uninitialized pointers begins with parameters of the entry function and all globally declared pointers (line 1). We do not restrict the seed set of pointers to `extern` variables (CUTE) as a routine may also rely on objects created within the module. If the global scope defines nested structs, pointer members of substructs are included recursively. Then a breadth-first search is performed until no new pointers are exhibited or a fixed depth bound is exceeded (line 4-13). The search follows the well known worklist pattern.

Some considerations complement the algorithm though they are not included in the above pseudo-code.

- In order to create a reasonable environment for pointers we face the problem that an `int *` may point to a single `int` or to an `int []`. We propose to decide whether to create an array or a single object dependent on a source code analysis that reports if the pointer is subject to any pointer arithmetic or index operation. In this case it seems reasonable to create an array of simple objects instead of a single one.
- If the depth bound terminates our algorithm, we must decide how the uninitialized pointers in the current worklist are treated. We suggest that these should be initialized to `null`.
- Common abstract data types in Linux can be created by detailed templates. The most prominent example is the definition of linked lists. List elements are then required to be structs with one field having the type `struct list_head` from `include/linux/list.h`. For each list implemented in this way, we suggest to create a bounded stub of this list with pointers pointing to the next list element. The last element terminates the list by a `null` next field. For other predefinied data types initialization can be accomplished in a similar fashion.
- Class invariants over primitive data types, e.g. sorted lists with a `int` key field, can be encoded by `assume` statements.
- The unrolling depth of each single data structure may be independently, non-deterministically chosen.

Using this strategy, aliasing occurs only when it is explicitely specified by a Linux abstract data type template. A short example of results produced by our algorithm is presented in Figure 2. For further aliasing between other arbitrary objects we offer a generic model in the next section.

```
struct cdrom_device_info_stub1;
struct cdrom_device_info_stub2;
struct cdrom_device_info_stub3;
...
void init_environment() {
// pointer initialization
   struct cdrom_device_info *
      parameter_stub = &cdrom_device_info_stub1;
   cdrom_device_info_stub1.next =
           &cdrom_device_info_stub2;
   cdrom_device_info_stub2.next =
           &cdrom_device_info_stub3;
// bound reached
   cdrom_device_info_stub3.next = null;
// call to entry point of module
   int parameter_stub2;
   open(parameter_stub,parameter_stub2);
}
```

Figure 2: Result of a manual execution of our algorithm.

3.1 Alias Modeling

In the above section we initialized all pointers by different objects. Though many device driver environments might be modeled successfully using this conservative alias policy, a finer-grained analysis might be included into our environments. A classical must / may alias analysis, or a user specified equivalent description, could specify which pointers may alias each other. These specifications may be implemented easily in a small initialization code block that can be automatically generated.

For each must alias analysis we insert a new assignment statement into the code block: we translate `p1` *must* alias `p2` into a statement `p1 = p2;` . For *may* alias relationships, we could exploit the modeling of non-deterministic data values in CBMC. The built-in CBMC function `nondet_bool()` returns either true or false. Hence we may easily translate may alias relations: `p1` *may* alias `p2` results in a statement

```
if (nondet_bool())  p2 = p1;
```

The aliasing may hence lead to an exponential number of different initialization paths.

4. SUMMARY

Many tools for model checking C code exist. Though these tools offer a sophisticated treatment of most features of the language, our finding is that they are not yet stand-alone solutions for the modular analysis of Linux device drivers. In most cases the tools have to be complemented by an extensive operating systems model or at least a test driver that invokes and initializes all necessary members in the driver. It has been described that the construction of environments needs considerable effort. In [9] the effort dominated the overall work.

The contribution of our paper is an algorithm to automatically construct simple data environments for device drivers. We have shown that these environments might improve the precision of CBMC when analysing Linux device drivers. Other solutions for interface generations exist, though they seem either dedicated to Java programs (e.g. [7]) or they aim to provide a general solution loosing the advantages the predefined Linux abstract data types. Our solution only needs a specified entry point for each module and templates for the few abstract data types in Linux.

Despite the expected advance by widening the application domain in CBMC to modular programs, we left several problems unsolved. The creation of the environment is heuristic and requires an external specification of possible alias relationships. The identification of abstract data types is heuristically inferred and must be checked and potentially corrected by the user. The same situation is given for the inference whether a pointer points to a single element or an element within an array.

Early prototypes result in hundreds of generated initialization code lines even for small list bounds of 3. CBMC was able to process the examples and find some known pointer errors that could not have been checked without our data environment.

5. REFERENCES

[1] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE)*, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.

[2] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[3] V. Contributors. The SLAM Project. http://research.microsoft.com/slam/, 2006.

[4] T. Henzinger, R. Jhala, R. Majumdar, and G. SUTRE. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN'2003), Portland, OR, USA, May 2003*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[5] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *Proc. of the 10th European Software Engineering Conference (ESEC/FSE-13)*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[6] D. van Leeuwen, E. Anderson, and J. Axboe. *A Linux Cdrom Standard*, kernel 2.6.15 edition, March 1999. Found in Documentation/cdrom.

[7] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, April 2005.

[8] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 351–363, New York, NY, USA, 2005. ACM Press.

[9] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 273–288. Springer Berlin / Heidelberg, 2004.