SLAyer: Memory Safety for Systems-Level Code

Josh Berdine, Byron Cook, and Samin Ishtiaq

Microsoft Research

Abstract. SLAyer is a program analysis tool designed to automatically prove memory safety of industrial systems code. In this paper we describe SLAyer's implementation, and its application to Windows device drivers. This paper accompanies the first release of SLAyer.

1 Introduction

This paper describes SLAyer, a program analysis tool designed to prove the absence of memory safety errors such as dangling pointer dereferences, double frees, and memory leaks. Towards this goal, SLAyer searches for invariants that form proofs in Separation Logic [8]. The algorithms implemented in SLAyer are aimed at verifying moderately sized (*e.g.* 10K-30K LOC) systems level code bases written in C. SLAyer is fully automatic and does not require annotations or hints from the user.

2 Example

The majority of Windows faults are caused by third-party device drivers [1]. Because device drivers spend much of their time maintaining queues of requests, many of these errors are related to maintaining memory safety while operating over mutable linked data structures. Consider the code excerpt from the FireWire device driver distributed in the Windows Driver Kit (WDK) v7600 shown in Figure 2. The code is part of the cleanup routine in which allocated "isoch" resources are deleted from the IsochResourceData list. The while loop (line 596) and if test (line 600) traverse the list.

The suspicious code is on line 604 where the element is removed from the wrong list. The code then assumes, on line 606, that listEntry is pointing into the middle of an ISOCH_RESOURCE_DATA , whereas it is actually pointing into a CROM_DATA . The assignment to IsochResourceData on this line now sets it to a parent object of the wrong type. SLAyer complains that it cannot verify that the later accesses to IsochResourceData are to valid memory. This is a real bug, now fixed in the Windows 8 codebase.

3 Applying SLAyer to Device Drivers

Although the current public release of SLAyer is as a standalone tool running on vanilla C code, we have also integrated it with Static Driver Verifier (SDV) [1].

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 178–183, 2011.

```
475  VOID
476  kmdf1394_EvtDeviceSelfManagedIoCleanup (
477                                          IN  WDFDEVICE Device)
478  {...}
496  {
497      PDEVICE_EXTENSION deviceExtension = NULL;
498      PLIST_ENTRY listEntry = NULL;
499  {...}
502      deviceExtension = GetDeviceContext(Device);
503  {...}
594      // Remove any isoch resource data
595      //
596      WHILE (TRUE)
597      {
598  {...}
600          if (!IsListEmpty(&deviceExtension->IsochResourceData))
601          {
602              PISOCH_RESOURCE_DATA    IsochResourceData = NULL;
603
604              listEntry = RemoveHeadList(&deviceExtension->CromData);
605
606              IsochResourceData = CONTAINING_RECORD (
607                  listEntry,
608                  ISOCH_RESOURCE_DATA,
609                  IsochResourceList);
610  {...}
676          } //  if (!IsListEmpty(&deviceExtension->IsochResourceData))
677  {...}
682      }
683  {...}
686  } // kmdf1394_EvtDeviceSelfManagedIoCleanup
```

Fig. 1. FireWire cleanup routine

In this integration, SLAyer is called instead of the model checker SLAM. The SDV OS model used is an extension to SDV's original OS model: it is developed to be faithful both to various I/O protocols as well as to the heap.

The top-level user interaction is to provide the C source code of a Windows Driver Framework (WDF) device driver, and get the result at the end of the run. The result is one of: Safe (a proof the target memory safety property was found); Possibly Unsafe (a failed proof in the form of an abstract counterexample providing a path to a memory safety violation); or, Exhausted (Time/Memory constraints exceeded). Figure 2 gives the overall tool flow picture.

A device driver consists of a set of dispatch routines. The OS model provides a `main` function that simulates the lifetime of a driver (calls the driver's dispatch routines) under the most general assumptions; it also provides behavioral specifications of the kernel functions that the driver calls. The Microsoft Visual C compiler based frontend links the driver with the OS model to form a complete, closed, sequential system that is the input to the SLAyer analyzer.

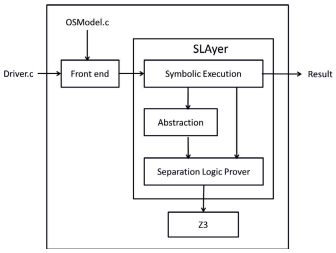

Fig. 2. SLAyer Flow

The OS model can be viewed as the assumption under which the memory safety property is proven for the source code. Additionally, the OS model imposes the obligation that the driver maintain data structure integrity for objects passed over the Driver–OS model interface. In this sense, underlying kernel and even hardware properties can be seen to leak into the driver code.

4 SLAyer Program Analysis

SLAyer implements an analysis that attempts to prove the absence of memory safety errors. Such an error occurs whenever dereferencing a pointer to an object outside the object's dynamic lifetime. Proving this property subsumes those such as double-frees, and null or dangling pointer dereferences.

If SLAyer finds a proof, then the input program under our semantics is memory safe. There are C programs (that overrun an array or access misaligned data, for example) that are safe in our idealized semantics, but have undefined meaning according to the C standard. This situation is characteristic of automatic "sound" tools.

To elaborate, the main idealization in our semantics is the model of memory: SLAyer takes a "logical", as opposed to a "physical", view of heap memory. Memory is modeled as a collection of disjoint structured objects. The structure of each object is determined by the source-level struct definition, omitting aspects such as alignment, padding, and byte widths of scalar values. The memory model is not even close to byte-accurate, so legitimate pointer traversals that use char* as a universal type are not provable.

The focus of SLAyer is reasoning about the shape of mutable linked data structures. To this end, validity of memory is treated at a per-object granularity. In particular, arrays and structs are treated as unbounded objects where accessing any member from a valid object is valid, and any index of a valid array is valid. This is deliberately in contrast to tools aimed at catching buffer overflows such as ESP [4].

4.1 Prover

Assertion Language. The particular fragment of Separation Logic used is, like other tools such as SpaceInvader [10] and Thor [9], an extension of that introduced in Smallfoot [3].

The assertion logic does not distinguish between the various C scalar types such as pointers, integers and floats. It does distinguish offsets, which are not first-class in C and represent differences between pointers to members of structured objects, and so are expressions that can be added to pointers via field access ".", or subtracted from them via CONTAINING_RECORD. Structured values are represented using a form of records mapping offsets to scalar values similar to a first-order theory of arrays (variables, select, store), but where the domain is given by an associated C struct definition.

The pure, heap-independent, part of the logic is essentially passed through to the Z3 SMT solver [5], thereby inheriting the same generality. Atomic pure formulas in particular include (principally linear) arithmetic ($<$, $\leq$), and equality over address expressions ($p = q$, $p.\mathsf{CromData} \neq p.\mathsf{IsochResourceData}$). Pure formulas are kept in negation-normal form.

Apart from Separation Logic's emp, which describes an empty part of the heap, atomic spatial formulas are of two forms: points-to or list-segment. A points-to $l \mapsto r$ describes a single heap cell at location l that contains an object

described by a record r. A list-segment $\mathsf{ls}(\Lambda, k, \boldsymbol{p}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{n})$ describes a possibly-empty, possibly-cyclic, segment of a doubly-linked list, where the heap structure of each item of the list is given by the formula Λ. This second-order inductive predicate is used to support complex composite data structures [2].

Formulas are closed under separating conjunction $P{*}Q$ and disjunction $P{\vee}Q$, and may have a prefix of existential quantifiers $\exists \boldsymbol{x}.Q$. Unlike related tools, formulas are not restricted to disjunctive-normal form, and arbitrary nesting of $*$ and $\vee$ is supported. While this generalization of representation does not add expressivity in principle, and is a substantial complication for the implementation, there are several motivations. One is the ability to compactly represent assertions that otherwise may blow up when expressed in DNF: for example it is common for different code branches to produce formulas whose heap structures differ in only a small region. Another is added flexibility in the design of theorem proving algorithms, where small proofs generally require (intermediate) formulas not in DNF.

Subtraction. All manipulation of formulas performed by the program analysis operations in SLAyer are, at the core, defined in terms of a judgment form called subtraction (a generalization of "frame inference" introduced in [3]), and implemented using a prover for these judgments. A subtraction judgment $M \vdash \exists \boldsymbol{x}.\, S \rightsquigarrow R$ holds if and only if the entailment $M \vdash \exists \boldsymbol{x}.\, (S{*}R)$ is universally valid. This is a production form, where a valid remainder R is computed as a function of the minuend M, existentials $\boldsymbol{x}$, and subtrahend S. Informally, a subtraction query $M \vdash \exists \boldsymbol{x}.\, S \rightsquigarrow$ asks the prover to re-express M, possibly weakening it, into the form $S * R$, thereby ensuring that heaps satisfying M have subheaps satisfying S, and yielding a formula R that describes the rest of each M heap.

Proof search is performed using a sequent calculus that includes deduction rules specific to the fragment's atomic formulas. A particular collection of axioms involving $\mapsto$ and ls are built into the calculus. These axioms generally require induction to prove, and by adding them to the prover it is able to prove inductive properties without searching for induction hypotheses. Additionally, these axioms, and knowledge of the semantics of the atomic forms, are used to direct aggressive use of Cut where subformulas of S are used to choose small but not atomic or quantifier-free intermediate proof goals. Searching for proofs with Cut enables localizing case analysis, resulting in much smaller proofs and search spaces, as well as more compactly represented remainders.

Reasoning about pure formulas is done using Z3, given an axiomatization of the pure fragment of the assertion logic. To enable incremental solving, during proof search SLAyer maintains a first-order approximation of the hypotheses in Z3. Leaves of proof trees are discharged by Z3, as they are implications between pure formulas. Z3 is also used to reason about equality between pointers in order to guide application of proof rules that manipulate spatial formulas. Additionally, some case splits in the sequent calculus are directed by unsatisfiable cores extracted by Z3. Overall this results in many small queries, involving the theories of arrays, data types, integer linear arithmetic, uninterpreted functions, as well as quantifier elimination for each.

The prover implemented in SLAyer is not complete, due (in part) to limited treatment of quantifiers and not attempting to find proofs using induction over the second-order list segments.

4.2 Symbolic Execution and Abstraction

Interprocedural Analysis. SLAyer uses a version of the Reps-Horowitz-Sagiv algorithm with localization [7] to perform a whole-program interprocedural analysis. The procedure specifications computed as summaries take the form $\forall \boldsymbol{g}.\{P\}f(\boldsymbol{x})\{Q\}$. The ghost variables $\boldsymbol{g}$ allow values on procedure entry to be compactly related to those on exit. Application of such specifications relies on subtraction's ability to reason adequately about quantified formulas. This parameterization is useful, for instance, to treat so-called heap cutpoints, as well as to express pure pre/post relations, to for instance reestablish properties of shadowed stack variables when returning from recursive procedures.

Transformers. Individual instructions, including specification statements, are symbolically executed following Smallfoot [3]. Given the generalization of subtraction to arbitrary disjunction and existential quantification, no separate "rearrangement" operation is needed in SLAyer.

Abstraction. The abstraction operation that generalizes formulas to loop invariants takes the form of a rewrite system that progressively weakens formulas, similar to SpaceInvader [6]. Instead of syntactic variable occurrence conditions, SLAyer uses rewrite rules guarded by properties involving a form of reachability through formulas modulo provable equality. Syntactic conditions proved too fragile when using a more general assertion logic. Another difference is that subtraction is used to perform the actual manipulation of the formulas, which means that the algorithms which determine how and if to rewrite do not need to know the full logical meaning of formulas.

Abstraction has also been extended to arbitrarily nested disjunction. When considering whether to apply a particular weakening of the formula, all the deeper disjuncts are considered, and distinctions where both alternatives appear are not preserved. This enables abstracting and hoisting common facts out of disjunctions, transforming formulas closer to conjunctive-normal form. So unlike the join operation of [10] which merges disjuncts of DNF formulas, the support of nested disjunction allows merging parts of formulas even when they cannot be fully merged.

5 Experimental Results and Availability

Table 1 presents some experimental results. The fw programs are extracted from the Windows FireWire driver, and are representative of device driver type code: a lot of control structures, traversal through linked lists, pointer arithmetic (the `cleanup_isochresourcedata` tests are the FireWire bug testcases). The sll programs are similar but avoid using CONTAINING_RECORD to factor out pointer

Table 1. Benchmarks

fw/attach_buffer_insert_head_list.c	Safe	1.8	sll/append.c	Safe	14.2
fw/callback_remove_entry_list.c	Safe	99.8	sll/copy.c	Safe	3.8
fw/cleanup_isochresourcedata.c	Safe	28.0	sll/copy_unsafe.c	Unsafe	0.3
fw/cleanup_isochresourcedata_unsafe.c	Unsafe	1.8	sll/create.c	Safe	0.1
fw/cromdata_add_remove.c	Safe	31.5	sll/create_kernel.c	Safe	3.8
fw/is_on_list_flat.c	Safe	18.2	sll/destroy.c	Safe	0.4
fw/is_on_list_via_devext.c	Safe	53.1	sll/filter.c	Safe	10.5
			sll/find.c	Safe	3.5
			sll/reverse.c	Safe	1.2
			sll/traverse.c	Safe	0.7

arithmetic. We are very conscious of the differences and prototype status of tools in this field; we present these results only as a strawman benchmark. The table gives the time, usr+sys in seconds, for SLAyer to prove each test Safe or Possibly Unsafe. The machine used was an Intel E5630 running Windows 7 64-bit.

SLAyer is available from `http://research.microsoft.com/slayer`. The download includes these benchmarks, and other C programs that stress memory safety. We plan to make a set of releases that revise the core components (analyzer performance, OS model fidelity), and to make our integration with SDV available.

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough Static Analysis of Device Drivers. In: EuroSys (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
4. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI (2002)
5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
7. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
8. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)
9. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: THOR: A tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
10. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)