Finding Bugs in Open Source Kernels using Parfait

Daniel Dawson, Nathan Hawes, Christian Hoermann,
Nathan Keynes, Cristina Cifuentes
Sun Microsystems Laboratories
Brisbane, Australia
{firstname.surname}@sun.com

ABSTRACT

Parfait is a static bug checking tool for C/C++ source code, which is designed to be both scalable and precise. Requirements for this tool were derived from interaction with the Solaris^{TM} operating system team, where it was required to check millions of lines of code in a time-efficient manner, with minimal noise and a low cost of integration into the build process.

This paper gives an overview of the Parfait tool and present the results of running Parfait over the OpenSolaris^{TM}, Linux and OpenBSD operating system kernels. It will also summarise the graphical reporting tool which helps developers quickly understand where bugs are in source code.

1. INTRODUCTION

Static bug checking tools are becoming increasingly sophisticated and use a variety of complex analysis techniques. Unlike earlier tools such as lint [11], the current generation of tools is able to analyse programs with an understanding of the language semantics, and detect specific real defects, rather than just potentially problematic constructs. As well as an increase in complexity, the increase in the processing capabilities available has made complex analysis techniques practical for everyday development. The goal of these tools is to find implementation defects at the earliest possible opportunity — as soon as the source code can be compiled and before the testing and integration phases. Bugs found prior to deployment are much cheaper to fix than those found after deployment, which in turn saves time and money for all concerned.

Static bug checking tools excel at finding implementation defects (e.g., null pointer dereferences) in rare or exceptional code paths that can be easily overlooked during typical testing. They have the additional advantage of being able to precisely report the lines of source code that would result in a runtime error, with no debugging required. However, they are by no means a silver bullet — no static analysis tool can locate all defects without producing some noise, nor can it find logical or behavioural defects without guidance. Some tools are able to detect inconsistencies between similar code fragments, but the tool cannot know for sure which version is correct without additional information. As such, bug checking tools are complementary to conventional testing and quality assurance processes.

There are three main requirements for a bug checking tool. Firstly, it must correctly report as many real bugs as possible. Secondly, it must minimize the number of false positives reported (that is, bug reports that are not real bugs). While all current tools will report at least some false positives due to limitations of the analyses, the number needs to be kept as low as possible. Finally, it must finish the analysis in a reasonable amount of time — tools that need a week or a day to run cannot be integrated into a nightly build, let alone the typical day-to-day development process. These requirements were the starting point for the Parfait design.

Kernel Conference Australia '09 Brisbane, Queensland, Australia
(c) 2009, Sun Microsystems, Inc.

2. RELATED WORK

In the last decade a variety of bug checking tools based on static analysis techniques have become available in academia and industry. These tools mainly support popular languages such as C, C++ and the Java^{TM} programming language, as well as a variety of different types of bugs, such as buffer overflows, cross-site scripting, SQL injection, deadlocks and race conditions. Tools vary in accuracy, performance and cost.

Popular commercial tools include Coverity Prevent [7], Klocwork Insight [12], Fortify 360 [8] and PolySpace [6]. PolySpace focuses on embedded real time systems code, while the other tools focus on general application code. Research and academic tools include FindBugs [10], a static bug checker for the Java^{TM}programming language; ASTRÉE [4], a verification tool for embedded systems code; and the LLVM Clang static analyzer [?] for Objective C and C programs.

In recent years Coverity has scanned open source software and reported bugs to the open source community [5]. The Linux kernel has been scanned with the Coverity Prevent tool several times and reported bugs have been fixed.

To our knowledge, no other open source kernel has been reported in the literature to be scanned with a bug checker. In this paper we report on results from scanning 3 open source kernels; the OpenSolaris ON consolidation, the Linux kernel and the OpenBSD kernel; with Parfait's static and dynamic buffer overflow analyses. Bugs found have or are being submitted to their respective communities.

3. THE PARFAIT DESIGN

The Parfait design addresses the requirements of recall, precision and scalability by defining an extensible framework composed of layers of program analyses [3]. Precision is the ratio of bugs that are reported correctly to bugs that are not. Scalability is the ability of the tool to produce results in a time efficient manner as larger codebases are analysed. Recall is the percentage of correctly-reported bugs over all possible bugs that exist in a given codebase.

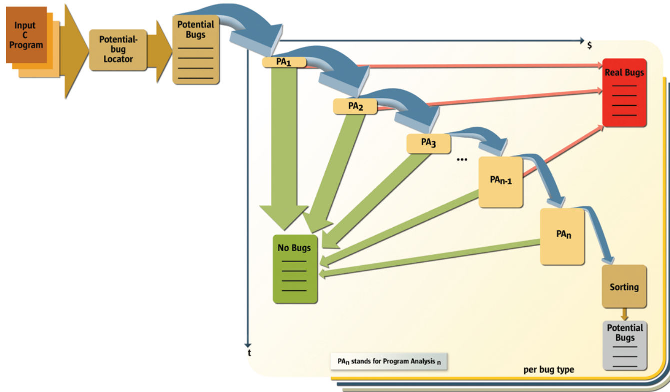

Figure 1: The Parfait Architecture

Figure 1 illustrates the Parfait architecture. For greater precision and scalability in bug checking, the Parfait design employs an ensemble of static program analyses that vary in complexity and expense. Program analyses (PA_i) are ordered from the least to most time expensive, ensuring that each buggy statement is detected with the cheapest program analysis capable of detecting it, effectively achieving higher precision with smaller runtime overheads.

For recall, Parfait compiles a list of potentially buggy statements per bug type for the framework to process. Each analysis can then determine one of three things about a potential buggy statement: it can confirm it is a bug, it can reject it as a non-bug, or it cannot tell either way. There are three lists attached to these three results: real bugs, no bugs, and potential bugs. After the ensemble of analyses is run, the user can see the real bugs list (which has high precision) and the potential bugs list (to get an idea of the recall rate; these results need to be manually verified by the user). In practice, users only process the real bugs list.

To overcome the computational program analysis bottleneck, y demand driven program analysis instead of traditional whole program analyses have been employed. The demand driven analyses effectively generate a slice of a program starting at each potential buggy statement, limiting the scope to a small and relevant subset of the code. This allows for the tool to be more scalable.

Layers of Program Analyses

Program analyses vary in power and complexity: a simple analysis can find a small subset of bugs very cheaply, whereas a more complex analysis can find a larger subset of bugs more expensively, normally being able to find the simpler bugs as well as the more complex ones, that is — there are obvious bugs and there are complex bugs. A cheap analysis can find the obvious bug but it cannot find the complex bug. A complex analysis can find both types of bugs, but it usually takes a relatively longer runtime to find them.

To detect read/write buffer overruns (i.e., buffer overflows and read outside array bounds), three analyses have been implemented in Parfait at present time: constant bounds checks, partial evaluation, and symbolic analysis – all of which are mainly intra-procedural.

For constant bounds checks, constant propagation and folding [1] are used to trivially check array references with constant indexes against valid array bounds.

For statements with statically-computable dependencies on an array index, partial evaluation [9] techniques are used to specialise a slice of code around the buggy statement with the constant inputs, and execute the slice in an interpreter to determine whether the index is within valid bounds or not.

For other statements symbolic analysis [2] is used to compute the symbolic range of variables. The symbolic range of an array index variable will then be compared with the array bounds to determine whether or not the array reference is out of bounds.

Full support for the C language requires support for the C library; an ISO standard that specifies well defined semantics for the various functions in the library [13]. In Parfait, some use of pre-conditions is employed to support checking of C library function semantics. Post-conditions will be added in the future.

4. TESTING OPEN SOURCE KERNELS

Throughout the implementation of Parfait, the OpenSolaris Operating System/Networking (ON) consolidation has been used as a proof-of-concept test. Having access to the developers of the code allowed the team to more easily determine the accuracy of the tool. Nightly runs of Parfait over the ON code allows us to assess scalability as new analyses are introduced or changed.

Testing Parfait with kernel code from the OpenSolaris, Linux and OpenBSD operating systems, and comparing the defects found in each of these kernels provides a benchmark for Parfait developers and an insight into the trends and similarities between the defects reported for each kernel. In the data presented herein, Parfait is only running read/write buffer overrun analyses.

The following sections provide results of running Parfait over the various kernels, and presents code fragments showing examples of bugs found by Parfait, as well as false positives being reported at the time of writing.

4.1 Kernel Results

Table 1 shows the evaluation of Parfait over the three open source kernels. For each kernel we show: the time it took Parfait to analyse the code (in minutes) on an AMD Opteron 2.8 GHz machine, the number of non-comment lines of code

Kernel	Time (min)	Core			Device Drivers		
		LOC	Buf Overruns	Bug Density	LOC	Buf Overruns	Bug Density
OpenSolaris UTS b105	5	2,147,884	15	0.0069	1,234,881	67	0.054
Linux 2.6.29	13	1,637,947	12	0.0073	4,146,313	85	0.020
OpenBSD 4.4	2	493,455	3	0.0060	869,548	26	0.029

Table 1: Parfait evaluation of open source kernels

(LOC) excluding non-x86 architecture-specific code that was not analysed, the number of read and write buffer overruns, and the bug density; which is computed as the rate of bugs found per 1000 lines of code (KLOC). The results are broken into two groups: the core kernel and device drivers. This is because device drivers are often written by third parties and as such standard code practices used in the rest of the kernel do not always apply. The number of bugs reported in the table are for those bugs that were manually verified as real bugs. The directories that were part of the kernel comparison for each of the distributions were: the `uts` directory for the OpenSolaris ON consolidation, the complete kernel distribution for Linux, and the `sys` directory for OpenBSD.

Of the three open source kernels tested, the Linux kernel has been checked with the Coverity Prevent tool in multiple years. It was surprising to us to find that many bugs in code we thought to be clean, however, the churn rate in the Linux community is higher than that in the other two communities.

Perhaps unsurprisingly, device drivers are disproportionately represented in bugs reported by Parfait. For OpenSolaris, 81% of bugs were reported against device driver modules (counting all `io` directories), which collectively accounts for 36% of the codebase. For Linux, 87% of bugs were found in driver code (`drivers` or `sound` directories), which accounts for 71% of the codebase. And for OpenBSD, devices drivers (`dev`) contribute 63% of the code size, and have 89% of the bugs found.

4.2 Bug Examples

Shown below are three examples of bugs found by Parfait, this illustrates the possible types of bugs that different buffer overflow analyses can locate in source code.

The following code from the OpenSolaris ON uts/i86xpv/.../io /pci/pci_tools.c file shows a sample buffer overrun detected by constant bounds checks:

```
231 static void
232 pcitool_get_intr_dev_info(dev_info_t *dip,
                              pcitool_intr_dev_t *devs)
233 {
234   (void) strncpy(devs->driver_name,
235      ddi_driver_name(dip), MAXMODCONFNAME-1);
236   devs->driver_name[MAXMODCONFNAME] = '\0';
237   (void) ddi_pathname(dip, devs->path);
238   devs->dev_inst = ddi_get_instance(dip);
239 }
```

Parfait reports the following error:

```
Error: Write outside array bounds at
   /zcratch/sunlabs/onnv-20080624/usr/src/uts/i86xpv/npe/../../
   i86pc/io/pci/pci_tools.c:236 in function
   'pcitool_get_intr_dev_info' [Constant index checks]
In array dereference of devs->driver_name[256] with index '256'
Array size is 256 bytes, index is 256
```

In the code, the array `devs->driver_name` is declared with size `MAXMODCONFNAME`. The write of the end-of-string character at line 236 is therefore off-by-one.

The following code from the Linux drivers/net/eexpress.c file shows a sample buffer overrun at line 1474 detected by the partial evaluation analysis:

```
1470 for (i = 0; i < (sizeof(start_code)); i+=32) {
1471    int j;
1472    outw(i, ioaddr + SM_PTR);
1473    for (j = 0; j < 16; j+=2)
1474      outw(start_code[(i+j)/2],
1475           ioaddr+0x4000+j);
1476      for (j = 0; j < 16; j+=2)
1477        outw(start_code[(i+j+16)/2],
1478             ioaddr+0x8000+j);
1479 }
```

Parfait reports the following error:

```
Error: Read buffer overflow at
   /usr/src/linux-2.6.29/drivers/net/eexpress.c:1474 in
   function 'eexp_hw_init586' [Partial evaluation]
In array dereference of start_code[((i + j) / 2)]
   with index '((i + j) / 2)'
Array size is 69 elements (of 2 bytes each), index is 69
```

Where the `start_code` array is a 69-element array of 2-byte integers. Executing the loop starting at line 1473 would cause several additional words to be read past the end of the array and written to the device. A similar error also gets reported on line 1477.

Last, the following code from the Linux drivers/scsi/eata.c file shows a sample off-by-one bug at line 1513 detected by symbolic analysis:

```
1508 int ints[MAX_INT_PARAM];
1509 char *cur = str;
1510 int i = 1;
1511
1512 while (cur && isdigit(*cur)
                && i <= MAX_INT_PARAM) {
1513   ints[i++] = simple_strtoul(cur, NULL, 0);
```

Parfait reports the following error:

```
Error: Buffer overflow at
   /usr/src/linux-2.6.29/drivers/scsi/eata.c:1513
```

```
   in function 'option_setup' [Symbolic analysis]
In array dereference of ints[i] with index 'i'
Array size is 10 elements (of 4 bytes each), index <= 10
```

The conditional of the `while` loop at line 1512 allows variable `i` to equal `MAX_INT_PARAM`, which causes an overflow at line 1513.

4.3 False Positive Examples

False positives are bugs reported by the tool that are not real bugs in practice. Two different false positives reported by Parfait are shown below.

The following false positive reported in the Linux drivers/isdn/hardware/eicon/diva.c file is due to the merging of the value ranges for `max` and `nr`. Individually, the maximum values are `MAX_ADAPTER` and `4` respectively, so the symbolic analysis currently derives the maximum value of `i+j` to be `MAX_ADAPTER + 4 - 2`, which would be out of bounds. Work is underway to improve path sensitivity of the analysis to correctly handle these cases.

```
switch (CardOrdinal) {
...
case CARDTYPE_DIVASRV_VOICE_Q_8M_V2_PCI:
  max = MAX_ADAPTER - 4;
  nr = 4;
  break;
default:
  max = MAX_ADAPTER;
  nr = 1;
}

for (i = 0; i < max; i++) {
  if (!diva_find_free_adapters(i, nr)) {
     ...
     for (j = 1; j < nr; j++) {
       pa = diva_q_get_next(&pa->link);
       if (pa && !pa->interface.cleanup_adapter_proc) {
           ...
           IoAdapters[i + j] = &pa->xdi_adapter; /* FP */
           ...
           diva_os_enter_spin_lock(&adapter_lock,
                                   &old_irql, "add card");
       } else {
           DBG_ERR(("slave adapter problem"))
           break;
       }
    }
    return (pdiva);
  }
}
```

And the following false positive in the OpenSolaris ON uts/common/inet/inetddi.c file is notable because its correctness depends on the values in the `netdev_privs` array, and specifically on the fact that the last row contains null values that are never changed (but are not declared constant). Without that knowledge, the analysis incorrectly assumes that variable `i` may be able to equal `9`, past the end of the `netdev_privs` array. This false positive points at the need for field-sensitive analysis.

```
static struct dev_priv {
   char *driver;
   int privonly;
   const char *read_priv;
   const char *write_priv;
} netdev_privs[] = {
   {"icmp", PRIVONLY_DEV,  ... PRIV_NET_ICMPACCESS},
   {"icmp6", PRIVONLY_DEV, ... PRIV_NET_ICMPACCESS},
   ...
   {"spdsock", PRIVONLY_DEV, ... PRIV_SYS_IP_CONFIG},
   {NULL,      0,            ... NULL}
};

static int
inet_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
  int i, ndevs;

  if (cmd != DDI_ATTACH)
    return (DDI_FAILURE);
  inet_dev_info = devi;
  ndevs = sizeof(netdev_privs) / sizeof(struct dev_priv);
  for (i = 0; i < ndevs; i++) {
    char *drv = netdev_privs[i].driver;
    if (drv == NULL ||
                 strcmp(drv, ddi_driver_name(devi)) == 0)
      break;
  }
  return (ddi_create_priv_minor_node(devi, INET_NAME,
    S_IFCHR, INET_DEVMINOR, DDI_PSEUDO,
    netdev_privs[i].privonly, netdev_privs[i].read_priv,
    netdev_privs[i].write_priv, INET_DEFAULT_PRIV_MODE));
}
```

5. THE PARFAIT UI

To assist developers and managers in understanding the output from Parfait, a web-based graphical reporting interface has been developed. The Parfait UI was primarily designed to perform two tasks: provide a broad overview of bug information, and provide a mechanism for investigating individual bugs to determine their cause.

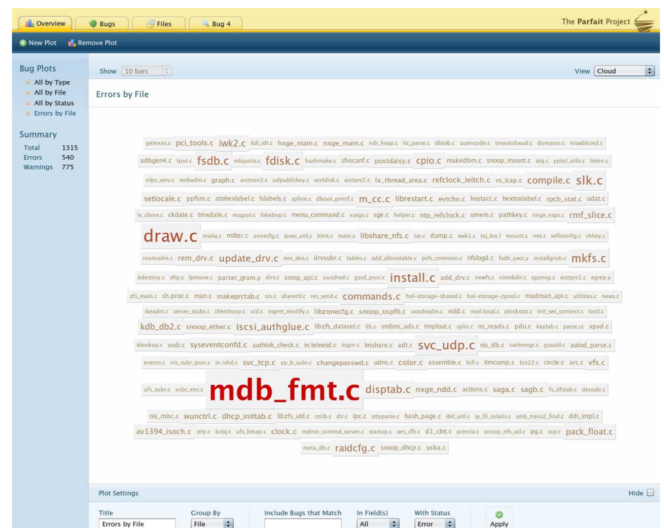

Figure 2: Tag cloud of files with bugs in the OpenSolaris ON consolidation codebase

In the current prototype of the UI, the first of the two primary tasks is largely effected with a plotting widget that summarises bug information. The plotting widget employs a three step methodology. Users first filter the bug list to obtain the subset they are interested in summarising, then choose the bug attribute by which they wish to categorise the data (e.g. location, type, status, etc.), and finally select whether to view the plot as a tag cloud or more traditional bar graph. This simple widget allows managers to

see where bugs are occurring, what types of bugs are occurring and how severe these bugs are. It uncovers defective regions in the codebase and any common, repeated errors. Figure 2 gives a snapshot of a tag cloud for the OpenSolaris ON consolidation bugs.

```
File source/usr/src/uts/i86xpv/npe/../../common/sys/pci_tools.h
141  /*
142   * PCITOOL_DEVICE_GET_INTR ioctl data structure to dump out the
143   * ino mapping information.
144   */
145
146  typedef struct pcitool_intr_dev {
147          uint32_t        dev_inst;       /* device instance - from kernel */
148          char            driver_name[MAXMODCONFNAME];    /* from kernel */
149          char            path[MAXPATHLEN]; /* device path - from kernel */
150  } pcitool_intr_dev_t;
151
152  typedef struct pcitool_intr_get {
153          uint16_t user_version;          /* Userland program version - to krnl */
154          uint16_t drvr_version;          /* Driver version - from kernel */
155          uint32_t        ino;            /* interrupt number - to kernel */

File source/usr/src/uts/i86xpv/npe/../../i86pc/io/pci/pci_tools.c
225                  rval = EFAULT;
226          return (rval);
227  }
228
229
230  /* It is assumed that dip != NULL */
231  static void
232  pcitool_get_intr_dev_info(dev_info_t *dip, pcitool_intr_dev_t *devs)
233  {
234          (void) strncpy(devs->driver_name,
235              ddi_driver_name(dip), MAXMODCONFNAME-1);
236          devs->driver_name[MAXMODCONFNAME] = '\0';
237          (void) ddi_pathname(dip, devs->path);
238          devs->dev_inst = ddi_get_instance(dip);
```

Figure 3: Fish-eye view of a bug along with dependencies across two separate files

The UI provides several tools to investigate individual bugs. Primary among them is a fish-eye view of the source code. For a particular bug users are shown a small contextual region of the source code centered on the bug's location. Surrounding this are other such regions centered on the various dependencies of the bug, such as variable declarations, associated structures and any relevant branch statements. This, combined with the bug message, makes determining the cause of the bug a quick and simple process. Figure 3 shows the fish-eye view of the bug at the line with the "bug" icon, highlighted in red, and gives dependencies for statements that were used by Parfait to determine this bug exists (such dependencies are in orange), along with the files where the dependencies exist.

Another useful widget is the source tree browser, which allows users to navigate through the directory structure to view an entire source file with all the bugs it contains highlighted and annotated with the bug information Parfait has produced. As the user navigates through the tree, each directory label is coloured to indicate how buggy it is relative to the other folders at the same depth in the tree. The absolute number of bugs is also displayed adjacent to it.

During development it was found that, this web-based UI has proven very useful to developers as it's simple and easy to use, and can quickly give users enough information for them to determine where the bug is and whether they agree with the tool that a reported bug is indeed a bug.

6. CONCLUSIONS

The number of buffer overrun bugs found by Parfait in mature open source kernels shows that static analysis tools are useful and have the potential to improve the quality of newly written and legacy code. Testing with the Linux, OpenBSD and OpenSolaris kernels has indicated that static analysis tools can find bugs even in mature, long-standing codebases.

Using Parfait in the development and maintenance phases of a project can benefit an organisation by improving software quality before shipment rather than relying on user feedback. This allows developers to focus on software improvement rather than maintaining bug reports which can be vague or incomplete, resulting in an improved overall software quality and reducing the time spent on maintenance. Further, fixing bugs before shipment is much cheaper than fixing them once the product has shipped; allowing cost savings to an organisation that makes use of these tools.

During development it was found that, coupling the tool with a web-based graphical reporting mechanism has helped significantly the use of the tool by developers in various different organisations.

Future Work

In order to make Parfait more relevant to kernel codebases, emphasis on kernel-specific problems need to be taken into account. For example, adding support for the differences between user addresses and kernel addresses, having a way to specify kernel-defined library support for use in the Parfait analyses (e.g., memory management functions defined by the kernel for its own use), and analyses for device driver bugs; being that many device drivers are third party and as such consistently have more bugs than any other part of the kernel codebase.

Acknowledgments

We would like to thank Scott Rotondo for providing feedback on behalf of the OpenSolaris ON consolidation team. Thanks also to David Gwynne for information and help with OpenBSD, and Lin Gao for helping to manually verify bugs reported by Parfait against the Linux codebase.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. Softw. Eng.*, 5(4):402–417, 1979.

[3] C. Cifuentes and B. Scholz. Parfait – designing a scalable bug checker. In *Proceedings of the ACM SIGPLAN Static Analysis Workshop*, pages 4–11, 12 June 2008.

[4] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *In Proceedings of the European Symposium on Programming*, number 3444 in Lecture

Notes in Computer Science, pages 21–30. Springer, April 2005.

[5] Coverity. Open source report 2008. scan.coverity.com, 2008.

[6] A. Deutsch. Static verification of dynamic properties. PolySpace White Paper, February 2004.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 23–25, San Diego, CA, Oct. 2000. USENIX.

[8] Fortify Static Code Analysis (SCA). `http://www.fortify.com/products/sca/`. Last accessed: 1 April 2008.

[9] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1971.

[10] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 92–106, Vancouver, BC, Canada, Oct. 2004. ACM Press.

[11] S. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1978.

[12] M. Webster. Leveraging static analysis for a multidimensional view of software quality and security: Klocwork's solution. White paper, IDC, Framingham, MA, Sept. 2005.

[13] WG14. *ISO C 99 Standard - TC2*. ISO/IEC Working Group 14, 9899:TC2 edition, May 2005.