Parfait: A Precise and Scalable Static Analysis Tool

Cristina Cifuentes
Architect & Research Director, Oracle Labs Australia
September 2012
About Oracle Labs
http://labs.oracle.com/

• Mission:
  – “Identify, explore, and transfer new technologies that have the potential to substantially improve Oracle’s business”

• Research portfolio
  – exploratory research
  – directed research
  – consulting
  – product incubation

• Locations
  – Based at Oracle HQ in Redwood Shores, CA
  – Research centres
    • Boston, MA; Austin, TX; San Diego, CA; Brisbane, Australia
  – Senior researchers around the world
About Oracle Labs, Australia
http://labs.oracle.com/projects/downunder/

- Based in Brisbane
- Focus on Program Analysis

```c
#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
    }
    buf[n] = '\0';

    n = atoi(argv[1]);
    buf2 = (char*)malloc(n);
    for (i = 0; i <= n; i++) {
        buf2[i] = argv[2][i];
    }
    return 0;
}
```

- Compile it
- Test it
- Understand it
- Verify it
- Bug-check it
- Security-check it
About Oracle Labs, Australia
http://labs.oracle.com/projects/downunder/

• Based in Brisbane
• Focus on Program Analysis

```c
#define BUFF_SIZE 100

int main (int argc, char *argv[])
{ char buf[BUFF_SIZE], *buf2;
  int n = BUFF_SIZE, i;

  for (i = 1; i <= n; i++) {
    buf[i] = 'A';
  }
  buf[n] = '\0';

  n = atoi(argv[1]);
  buf2 = (char*)malloc(n);
  for (i = 0; i <= n; i++) {
    buf2[i] = argv[2][i];
  }
  return 0;
}
```
Static Code Analysis Tools
Bugs are Part of Life
# Approaches to Bug-Checking

## DYNAMIC
- Instrumentation of code
- Runtime checks
- Inability to effectively collaborate—both internally and externally
- Small code coverage (approx. 10%)
- 10-300x slowdown
- Manual product issue resolution processes
- Detection of bugs is postmortem

## STATIC
- Analysis of source code
- Complexity issues of the analysis
- Good code coverage
- Imprecision of the analysis
- Detection of bugs before executing program and before deployment
Sample Static Code Analysis Tools
### Techniques used in Static Code Analysis Tools

<table>
<thead>
<tr>
<th>TOOL</th>
<th>LANGUAGE</th>
<th>TYPE OF ANALYSIS</th>
<th>INSTITUTION</th>
<th>PUBL. YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC</td>
<td>M3, Java</td>
<td>theorem proving</td>
<td>DEC WRL</td>
<td>1998</td>
</tr>
<tr>
<td>JPF</td>
<td>Java</td>
<td>model checking</td>
<td>NASA</td>
<td>1999</td>
</tr>
<tr>
<td>PREfix</td>
<td>C, C++</td>
<td>data flow, abstract interpretation</td>
<td>Intrinsa, Microsoft</td>
<td>2000</td>
</tr>
<tr>
<td>Jlint</td>
<td>Java</td>
<td>data flow, abstract interpretation</td>
<td>Knizhnik, Artho</td>
<td>2001, 2004</td>
</tr>
<tr>
<td>SLAM</td>
<td>C, C++</td>
<td>model checking, abstract interpretation, theorem proving</td>
<td>Microsoft RL</td>
<td>2001, 2002</td>
</tr>
<tr>
<td>ESP</td>
<td>C</td>
<td>data flow</td>
<td>Microsoft CSE</td>
<td>2002</td>
</tr>
<tr>
<td>PREfast</td>
<td>C, C++</td>
<td>data flow</td>
<td>Microsoft RL</td>
<td>2002</td>
</tr>
<tr>
<td>Splint</td>
<td>C</td>
<td>data flow</td>
<td>Univ of Virginia</td>
<td>2002</td>
</tr>
</tbody>
</table>
## Techniques used in Static Code Analysis Tools

<table>
<thead>
<tr>
<th>TOOL</th>
<th>LANGUAGE</th>
<th>TYPE OF ANALYSIS</th>
<th>INSTITUTION</th>
<th>PUBL. YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archer</td>
<td>C</td>
<td>abstract interpretation</td>
<td>Stanford</td>
<td>2003</td>
</tr>
<tr>
<td>PolySpace</td>
<td>Ada, C, C++</td>
<td>abstract interpretation</td>
<td>PolySpace</td>
<td>2004</td>
</tr>
<tr>
<td>FindBugs</td>
<td>Java</td>
<td>pattern matching</td>
<td>Univ. Maryland</td>
<td>2004</td>
</tr>
<tr>
<td>Blast</td>
<td>C</td>
<td>model checking</td>
<td>Berkeley</td>
<td>2004</td>
</tr>
<tr>
<td>K7</td>
<td>C, C++, Java</td>
<td>data flow</td>
<td>KlocWork</td>
<td>2005</td>
</tr>
<tr>
<td>SCA</td>
<td>C, C++</td>
<td>data flow</td>
<td>Fortify</td>
<td>2005</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>C, C++</td>
<td>abstract interpretation</td>
<td>GrammaTech</td>
<td>2006</td>
</tr>
<tr>
<td>Veracode</td>
<td>C, C++, Java</td>
<td>data flow</td>
<td>Veracode</td>
<td>2007</td>
</tr>
<tr>
<td>Clang Analyzer</td>
<td>C, Objective C, C</td>
<td>data flow</td>
<td>Apple</td>
<td>2008</td>
</tr>
</tbody>
</table>
Static Analysis Terminology

• Imprecisions
  – False positive
    • bug report that is not a real bug
  – False negative
    • bug missed by a tool
Static Analysis Terminology

• Imprecisions
  – False positive
    • bug report that is not a real bug
  – False negative
    • bug missed by a tool

• Properties
  – Soundness
    • a bug-checker is said to be sound if every bug in the source code being analysed is reported
  – Completeness
    • a bug-checker is said to be complete if every reported error is a real bug
Evaluation of Static Code Analysis Tools

• 2007, Sun Microsystems
  – Take long to run over millions of lines of code
  – Report too many false positives
Evaluation of Static Code Analysis Tools

• 2007, Sun Microsystems
  – Take long to run over millions of lines of code
  – Report too many false positives

• 2010, Forrester
  – Take long to run
  – Hard to use; need an expert user
  – False positive rate is high
The Parfait Design and Implementation
The Parfait Static Code Analysis Tool
Design Goals

• Scalability and precision
  – run fast
  – small false positive rate
The Parfait Static Code Analysis Tool

Design Goals

• Scalability and precision
  – run fast
  – small false positive rate

• Ease of use
  – integration into build process
  – support for understanding reported bugs
Parfait’s Architecture

Layered analyses:
- ordered, cheap to expensive
- sound analyses
- demand-driven

PA_n = Program Analysis_n
The Parfait Process

1- **Build** using the Parfait Compiler Wrappers
2- **Analyse** using the Parfait Static Code Analyser
3- **View** results using the Parfait Server, HTML Report or Emacs
aSSA: Our Extension to SSA Form

• SSA form
  – At join points introduces
    \[ y = \Phi(y_1,\ldots,y_k) \]
    where \( y_1,\ldots,y_k \) are different possible values of variable \( y \)

• aSSA form
  – At join points introduces
    \[ y = \Phi'(y_1,\ldots,y_k; p_1,\ldots,p_l) \]
    where \( y_1,\ldots,y_k \) are different possible values of variable \( y \), and
    \( p_1,\ldots,p_l \) are control predicates of the definitions
n = getInput()

if (n < 100)

  m1 = 2 * n
  m2 = n + 1

  m3 = $\Phi'(m_1, m_2; (n < 100))$
Program Analyses via Buffer Overflow Examples
Constant Propagation

“Constant propagation is defined as the process to discover values that are constant on all possible executions of a program and to propagate these values through the program.”

K. Olmos and E. Visser
Strategies for Source-to-Source Constant Propagation
WRS 2002
Constant Propagation Example

static void debug_print_key_and_data(
    FILE *fp, char *str1, char *str2, char *key,
    int ksize, char *data, int dsize)
{
    ...  
    if (ksize > NFS_FHMAXDATA) {
        linkinfo_ent inf;
        (void) memcpy(&inf, data, sizeof (linkinfo_ent));
        debug_print_linkinfo(fp, &inf);
    } else if (ksize == NFS_FHMAXDATA) {
        fhlist_ent inf;
        (void) memcpy(&inf, data, sizeof (linkinfo_ent));
        debug_print_fhlist(fp, &inf);
    } else {    ...  }
}
**Constant Propagation Example**

`cmd/fs.d/nfs/nfslog/dctab.c`

```c
static void debug_print_key_and_data(
    FILE *fp, char *str1, char *str2, char *key,
    int ksize, char *data, int dsize)
{
    ...  
    if (ksize > NFS_FHMAXDATA) {
        linkinfo_ent inf;
        (void) memcpy(&inf, data, 3172);
        debug_print_linkinfo(fp, &inf);
    } else if (ksize == NFS_FHMAXDATA) {
        fhlist_ent inf;
        (void) memcpy(&inf, data, 3172);
        debug_print_fhlist(fp, &inf);
    } else { ... }
}
```
static void debug_print_key_and_data(
    FILE *fp, char *str1, char *str2, char *key,
    int ksize, char *data, int dsize)
{
    ...}
    if (ksize > NFS_FHMAXDATA) {
        linkinfo_ent inf;
        (void) memcpy(&inf, data, 3172);  // sizeof(inf) = 3172
        debug_print_linkinfo(fp, &inf);
    } else if (ksize == NFS_FHMAXDATA) {
        fhlist_ent inf;
        (void) memcpy(&inf, data, 3172);  // sizeof(inf) = 1100
        debug_print_fhlist(fp, &inf);
    } else { ... }
}
static void debug_print_key_and_data(
    FILE *fp, char *str1, char *str2, char *key,
    int ksize, char *data, int dsize)
{
    ...   
    if (ksize > NFS_FHMAXDATA) {
        linkinfo_ent inf;
        (void) memcpy(&inf, data, 3172);  // sizeof(inf) = 3172
        debug_print_linkinfo(fp, &inf);
    } else if (ksize == NFS_FHMAXDATA) {
        fhlist_ent inf;
        (void) memcpy(&inf, data, 3172);  // sizeof(inf) = 1100
        debug_print_fhlist(fp, &inf);
    } else { ... }
}
Constant Bounds Checking Stats
OpenSolaris b105 Code (9 MLOC)

- # locations with array dereferences: 756,000
- # constant array indexes: 33%
- # constant array and constant index: 17%
Partial Evaluation

“Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input. In essence, partial evaluation removes layers of interpretation.”

Consel C, Danvy O
Tutorial Notes on Partial Evaluation
POPL 1993
Partial Evaluation Algorithm in Parfait

- Determine if dependencies on array index are computable statically
- If so, determine the slice of statements that it depends on
  - annotate each basic block:
    \[ \text{isEmpty}(bb) = \text{true if } bb \text{ is in the slice, else false} \]
- **Forward-flow, all-paths DFA over the slice**
  - \( \text{In}(bb) = \bigcap_{p \in \text{pred}(bb)} \text{Out}(p) \)
  - \( \text{Out}(bb) = \text{In}(bb) \), if isEmpty(bb)
  - \( \text{Out}(bb) = bb \), otherwise
Partial Evaluation Code Generation in Parfait

• Slice extension
  – If outside array bounds, returns True, else returns False at end of loop

• Bail-out
  – Loop iteration > 3x size of the array
  – Could be done based on time

• Resulting code is executed on
  – Mixed interpreter/JIT
Two Commons Shapes of Graphs
Partial Evaluation Example

**cmd/lp/filter/postscript/postdaisy/postdaisy.c**

postdaisy.h:
```c
#define ROWS  400
#define COLUMNS 200
...
```

postdaisy.c:
```c
#include "postdaisy.h"
char    htabstops[COLUMNS];             /* horizontal */
char    vtabstops[ROWS];                    /* and vertical tabs */
...
static void inittabs(void)
{
    int i;
    for ( i = 0; i < ROWS; i++ )
        htabstops[i] = ((i % 8) == 0) ? ON : OFF;
    for ( i = 0; i < COLUMNS; i++ )
        vtabstops[i] = ((i * ovmi) > BOTTOMMARGIN) ? ON : OFF;
}
```
Partial Evaluation Example

cmd/lp/filter/postscript/postdaisy/postdaisy.c

postdaisy.c:

#include "postdaisy.h"
char    htabstops[200];                /* horizontal */
char    vtabstops[400];                /* and vertical tabs */
...
static void inittabs(void)
{
    int i;
    for ( i = 0; i < 400; i++ )
        htabstops[i] = ((i % 8) == 0) ? ON : OFF;
    for ( i = 0; i < 200; i++ )
        vtabstops[i] = ((i * ovmi) > BOTTOMMARGIN) ? ON : OFF;
}
Partial Evaluation Example

cmd/lp/filter/postscript/postdaisy/postdaisy.c

```c
for ( i = 0; i < 400; i++ ) {
    // htabstops[i] = ((i % 8) == 0) ? ON : OFF;
}

for ( i = 0; i < 200; i++ ) {
    // vtabstops[i] = ((i * ovmi) > BOTTOMMARGIN) ? ON : OFF;
}
```
Partial Evaluation Example

cmd/lp/filter/postscript/postdaisy/postdaisy.c

```c
for ( i = 0; i < 400; i++ ) {
    // htabstops[i] = ((i % 8) == 0) ? ON : OFF;
    if (i > 200)
        return true;
}
return false;

for ( i = 0; i < 200; i++ ) {
    // vtabstops[i] = ((i * ovmi) > BOTTOMMARGIN) ? ON : OFF;
    if (i > 400)
        return true;
}
return false;
```
Partial Evaluation Example

```
postdaisy.c:
#include "postdaisy.h"
char htabstops[200]; /* horizontal */
char vtabstops[400]; /* and vertical tabs */
...
static void inittabs(void)
{
    int i;
    for ( i = 0; i < 400; i++ )
        htabstops[i] = ((i % 8) == 0) ? ON : OFF;
    for ( i = 0; i < 200; i++ )
        vtabstops[i] = ((i * ovmi) > BOTTOMMARGIN) ? ON : OFF;
}
```
Partial Evaluation Stats
OpenSolaris b105 Code (9 MLOC)

• Average slice size = 16 statements
• Smallest slice = 8
• Largest slice = 70
• Our overhead
  – 4 statements exit condition (cmp, br, ret T, ret F)
  – 3 statements for bail-out conditions (PHI, test, branch)

• Results over 5,814 slices
Symbolic Analysis

“A form of static analysis in which symbolic expressions are used to denote the values of program variables and computations.”

T. E. Cheatham et al.
IEEE Transactions on Software Engineering
1979
Buffer Overflow Detection using Symbolic Analysis

• Symbolic Analysis
  – uses symbolic expressions to denote the values of program variables and computations

• E.g.:
  – an integer variable “v” can be represented by the symbolic range $S(v)=[\text{min, max}]$
    • int v;       // $S(v) = [v, v]$
    • v = 5;       // $S(v) = [5, 5]$
    • v = a*3;     // $S(v) = S(a) \times 3$
Symbolic Analysis in Parfait

- Uses control predicates to refine the symbolic range of a variable at its use

\[
\begin{align*}
n & = \text{getInput()} & \text{S}[n] = [n, n] \\
\text{if (n < 100)} & \\
\text{True} & \quad m_1 = 2 \times n & \text{S}[m_1] = [2^n, 200) \\
\text{False} & \quad m_2 = n + 1 & \text{S}[m_2] = (101, n+1]
\end{align*}
\]

\[
m_3 = \Phi^\prime (m_1, m_2; (n < 100))
\]
Symbolic Analysis Example

cmd/xntpd/xntpd/ntp_refclock.c

```c
/* sntp.h: */

#define REFCLK_MAX 32 /* maximum index (room to expand) */

/* ntp_refclock.c: */

#define MAXUNIT 4 /* max units */

static struct peer *typeunit[REFCLK_MAX + 1][MAXUNIT];

...

int refclock_newpeer(struct peer *peer)
{
    u_char clktpe;  int unit;

    clktpe = (u_char)REFCLOCKTYPE(&peer->srcadr);
    unit = REFCLOCKUNIT(&peer->srcadr);
    if (clktpe >= num_refclock_conf || unit > MAXUNIT ||
        refclock_conf[clktpe]->clock_start == noentry) {
        return (0);
    }
    ...
    memset((char *)pp, 0, sizeof(struct refclockproc));
    typeunit[clktpe][unit] = peer;
    ...
}
```
Symbolic Analysis Example
cmd/xntpd/xntpd/ntp_refclock.c

```c
static struct peer *typeunit[33][4];
...
int refclock_newpeer(struct peer *peer)
{
    u_char clktype; int unit;
    clktype = (u_char)REFCLOCKTYPE(&peer->srcadr);
    unit = REFCLOCKUNIT(&peer->srcadr);
    if (clktype >= num_refclock_conf || unit > 4 ||
        refclock_conf[clktype]->clock_start == noentry) {
        return (0);
    }
    ...
    memset((char *)pp, 0, sizeof(struct refclockproc));
typeunit[clktype][unit] = peer;
    ...
}
```
Symbolic Analysis Example
cmd/xntpd/xntpd/ntp_refclock.c

static struct peer *typeunit[33][4];
...

int refclock_newpeer(struct peer *peer)
{
    u_char clktype;  int unit;  // S[unit]=[unit,unit]

    clktype = (u_char)REFCLOCKTYPE(&peer->srcadr);
    unit = REFCLOCKUNIT(&peer->srcadr);  // S[unit]=[unit,unit]
    if (clktype >= num_refclock_conf || unit > 4 ||
        refclock_conf[clktype]->clock_start == noentry) {
        ...
        return (0);  // S[unit]=[5,unit]
    }
    ...  // S[unit]=[unit,4]

    memset((char *)pp, 0, sizeof(struct refclockproc));
    typeunit[clktype][unit] = peer;  // typeunit[clktype][0..4]
    ...
}
static struct peer *typeunit[33][4];
...

int refclock_newpeer(struct peer *peer)
{
    u_char clktype;  int unit;

    clktype = (u_char)REFCLOCKTYPE(&peer->srcadr);
    unit = REFCLOCKUNIT(&peer->srcadr);
    if (clktype >= num_refclock_conf || unit > 4 ||
        refclock_conf[clktype]->clock_start == noentry) {
        ...
        return (0);
    }
    ...
    memset((char *)pp, 0, sizeof(struct refclockproc));
    typeunit[clktype][unit] = peer;
    ...
}
Symbolic Analysis Stats
OpenSolaris b105 Code (9 MLOC)

• 605,560 out of 1,108,836 load/store instructions cannot be resolved by symbolic analysis
• 45% of array bounds are computable via symbolic analysis (back in 2009)
Symbolic Analysis

• Our analysis
  – is demand driven
  – uses simple symbolic representation
  – uses GSA representation
  – is path sensitive

• The key to speedup is determining a relevant subset of dependencies
Other Analyses

• Data flow analysis
  – demand-driven
  – finds memory leaks, use after free, double free, ...

• Taint analysis
  – global, forward-flow & global, summary-based
  – finds tainted data

• Points-to analysis
  – new formulation, an order of magnitude faster
  – finds points-to sets

• Model-based analysis
  – based on model checking, two orders of magnitude faster
  – tested with memory leak
Evaluation and Results
Parfait Evaluation
Normalised Data

MLOC = Million lines of non-commented code
Deployments throughout Internal Organisations

• Commit time
  – Parfait is run before new code is committed to the tree
  – code needs to be Parfait-clean before commit is accepted

• Build time
  – Parfait is run in the nightly
  – new bugs are reported to engineers the next day (integration into the bug tracking system)

• Ad hoc
  – Parfait is run by Release Engineering or QA
Platforms and Compilers Supported

- **Platforms**
  - Linux x86, x64
  - Solaris x86, SPARC
  - OS X
  - Windows (alpha)

- **Compilers**
  - Intel ICC C/C++ 10.x
  - SunStudio 12
  - GCC 3.x, 4.x
  - MSVC (alpha)
Parfait Server GUI: View Bug Data by Bug Type
Parfait Server GUI: Drill Down to Bugs
Conclusions
Conclusions

- Parfait is an in-house static code analysis tool
  - it’s fast
  - it’s precise
  - it’s easy to use
  - it’s easy to integrate into the build process

- Parfait can be customised
  - find bugs that are specific to your codebase

- Integration of Parfait into the development process allows for bugs to be found & fixed early in the development cycle
Q&A

http://labs.oracle.com/projects/parfait/
Hardware and Software

Engineered to Work Together