Catching Software Bugs Early at Build Time

An Overview of
SPARROW's Static Program Analysis Technology

July 2007

Contents

1 Introduction

SPARROW is a state-of-the-art static source code analyzer that points to fatal bugs in C and C++ source.

SPARROW is a well-founded static tool. SPARROW's analysis engine is created by our innovative semantics-based static analysis technology. SPARROW does not compile, simulate, nor execute the source. SPARROW just reads in your source, analyzes its semantics in its advanced mathematical model, and discovers, if any, bugs in the source.

1.1 SPARROW's Value

- **Find Bugs Before Testing**

  SPARROW analyzes C and C++ source and points to places of fatal flaws. Neither binary generation, testing, nor simulation is necessary.

- **More Bugs Than Others**

  SPARROW finds more bugs than other tools in the market, thanks to its deep semantic analysis. SPARROW understands all constructs of C and C++, hence can capture tricky side-effects (such as aliasing) of any C and C++ command.

  SPARROW is also equipped with an automatic classification technology that ranks alarms so that the user should see more probable alarms earlier.

- **All Automatic**

  SPARROW is a one-button solution. Just let SPARROW know of your source path. SPARROW understands your build system. No change to your code or build scripts is necessary.

- **Right After Your Source Is Ready**

  SPARROW analyzes just your software source, even before the software's whole source is ready. Because SPARROW does not execute your code, you don't have to prepare its execution environment.

1.2 QnA

Q: What is SPARROW?

A: SPARROW is a static source code analyzer that automatically detects fatal memory errors(memory leak & buffer overrun) in C and C++ source code without actually executing the source.

Q: Who needs it?

A: Software developers and quality assurance teams who want to reduce the cost of software errors. SPARROW reduces the high cost of late detection of fatal flaws in C and C++ programs.

SPARROW is particularly useful for embedded software, mission-critical software (in defense, automobile, aerospace and etc.), or other hard-to-test quality-sensitive software.

Q: Is SPARROW tailored only for some domain-specific C and C++ programs?

A: No. SPARROW effectively analyzes any C and C++ program.

Q: How is SPARROW different from testing?

A: Sparrow detects bugs without executing your source code, and Sparrow takes into account all execution scenarios of the code.

On the contrary, testing has to execute your code, delaying bug-detection until the running environment is ready. Testing cannot cover all inputs to your code because input combinations are usually infinite or too many.

Q: What is Sparrow's core technology?

A: Semantic-based static program analysis. Sparrow checks, with an engineered approximation, all the execution behaviors of the input program without executing it. The process is all automatic. The input to Sparrow is the target program's source code.

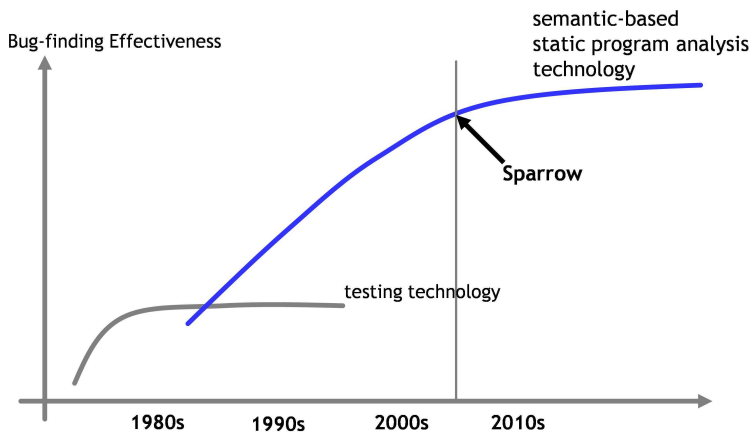


Figure 1: Technology Leaps by Semantic-based Static Program Analysis

As illustrated in Figure 1, it is semantic-based static analysis technology that has enabled an innovative leap forward to software quality assurance. Our research laboratory in academia has pioneered a research on this technology for more than 15 years.

Q: Is there any limitation or a hidden cost?

A: The user has to check if each reported alarm is a real bug or a false positive. Sparrow minimizes false alarms and its user interface is specialized to help the user's verification step.

2 Static Program Analysis Tools

2.1 Technology Spectrum in Static Analysis

A wide spectrum of technologies underpins the existing static tools in the market. One end are "syntactic" (also known as "pattern matching") tools. They emphasize on finding shallow bugs with small cost. Opposite end are "semantic" tools. They follow in varied degrees the run-time semantics of the input source.

It is semantic approach that finds most of hard-to-find bugs. Yet, we have to pay an increased analysis cost. As illustrated in the following figure, even within the semantic approach though, differs the balance between the analysis cost and the depth of the bug-finding coverage.

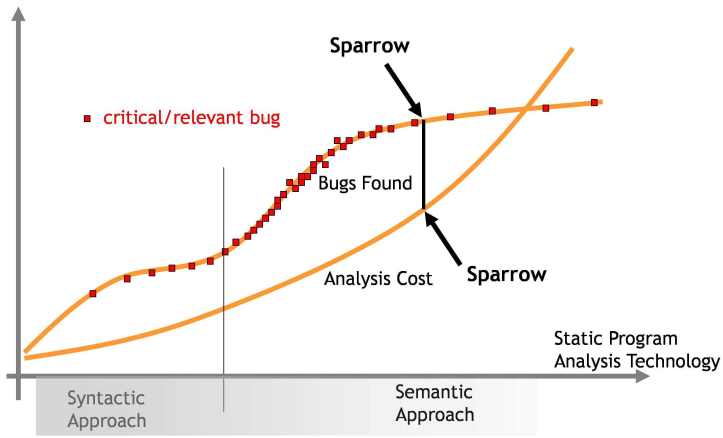


Figure 2: The Spectrum of Static Program Analysis Technologies

2.2 SPARROW's Competitive Positioning

Deep Analysis in a Cost-Effective Manner

In terms of Figure 2, SPARROW stands at the beginning of the second plateau of the "Bugs Found" curve. SPARROW hits right before the steep cost-increase point yet capable of finding most bugs in your code.

In other words, SPARROW's analysis engine strikes a careful balance between its analysis cost and its coverage of discovering targeted bugs. This balance has been achieved by our firm-founded semantic technology that has been tempered by extensive testing against a wide range of real-world, open-source and proprietary C and C++ software.

- SPARROW **analyzes deep.** SPARROW's deep semantic analysis finds bugs that other tools miss. SPARROW traces the input program's all execution scenarios yet in an economical way. This tracing, static analysis process is based on the semantics of the C and C++ languages. Example cases that demonstrates SPARROW's deep analysis capabilities are shown in Section 3.1.

  Some tools may be faster than SPARROW but only with shortcomings.

  - Such tools are effective in a limited way. They find bugs only if the bugs are among their trained set. If the set of trained bug patterns is different from your code's bug patterns, they fail to spot your bugs. It is analogous that they add numbers by looking up the addition table. If the input numbers are not in the addition table, they fail to add.
  - Such tools have a hidden cost. They may find bugs but has little basis to claim about the overall quality of the analyzed code. If they found only

one bug in your source, they can hardly claim that rest of your source has no bugs. They can only say that "your code has one bug in our bug patterns."

SPARROW's technology differs to be more considerate. Suppose SPARROW finds only one memory-leak position in your code. Though SPARROW may take some time to find the single leak, we can claim about your code's quality that the discovered leak is very likely the only one leak in your code. This likelihood is because SPARROW spends time to follow all possible execution scenarios, to be more exhaustive than other tools, rather than focusing on discovering particular patterns.

- SPARROW **is scalable.** SPARROW has virtually no limitation on the size of the input code. So far, SPARROW could analyze up to 10 million lines of C and C++ code at once. SPARROW achieves this scalability by analyzing the source files one by one, respecting the dependencies between them. SPARROW utilizes the disc space in order to always secure the main memory space while it analyzes an arbitrary number of large source files.

- SPARROW **aligns with the future.** SPARROW's semantic analysis technology is aligned with the evolution line of the future static bug-finding technology: verification.

  SPARROW's firm theoretical foundation is general enough to be easily instantiated into a "verification"-level analysis. SPARROW is based on a clear design specification which has been systematically derived from C and C++'s formal semantic definitions.

  SPARROW positions you to be smoothly migrated into the future, as we swiftly and continually offer additional advanced solutions along the technology evolution line.

3 Overview on SPARROW

SPARROW's analysis engine has been carefully designed based on necessary foundations in theory and sufficient tunings in practice.

SPARROW finitely computes the dynamics of programs at compile-time. Given as input a program source, it captures the execution semantics of the input source by a set of finite equations over an abstract space. SPARROW's process of setting up equations and computing their solutions is basically equivalent to tracing all the execution paths of the program, yet in an economical way.

3.1 Deep Semantic Analysis

SPARROW understands all constructs of C and C++, hence can capture tricky side-effects (such as aliasing) of any C and C++ command. In particular, SPARROW can analyze features such as:

· deep call chains	· pointer aliases	· dynamic memory allocations
· recursions	· infinite loops	· dynamic method bindings
· complex heap structures	· function pointers	· libraries

- SPARROW understands loop-induction variables and keeps track of their states throughout their scope. For example, following code is from `Linux Kernel 2.6.4` in `cdc-acm.c` where SPARROW discovered a buffer-overrun error.

```
570    static int acm_probe (struct usb_interface *intf,
571                          const struct usb_device_id *id)
572    {
...        ...
625        for (minor = 0; minor < 32 && acm_table[minor]; minor++);
...        ...
713        acm_table[minor] = acm;
714        usb_set_intfdata (intf, acm);
...        ...
719    }
```

SPARROW understands that the for-loop's induction variable `minor` has 32 after the loop. SPARROW keeps track of this `minor` value, understands it remains as 32 for next 88 lines of code until it overruns the buffer `acm_table` of size 32.

- SPARROW understands pointer flows across procedures and loops. Following example is from `tar-1.13` in `rmt.c` where SPARROW discovers a buffer overrun error.

```
045    #define STRING_SIZE 64
...    ...
125    static void
126    get_string (char *string)
127    {
128        int counter;
129
130        for (counter = 0; counter < STRING_SIZE; counter++)
131        {
132            if (safe_read (STDIN_FILENO, string + counter, 1) != 1)
133                exit (EXIT_SUCCESS);
134
135            if (string[counter] == '\n')
136                break;
137        }
138        string[counter] = '\0';
139    }
...    ...
182    int
183    main (int argc, char *const *argv)
184    {
...        ...
217        switch (command)
218        {
...            ...
221            case 'O':
222            {
223                char device_string[STRING_SIZE];
224                char mode_string[STRING_SIZE];
225
226                get_string (device_string);
227                get_string (mode_string);
```

From the `main` procedure, SPARROW understands that the calls to `get_string` at lines 226 and 227 pass pointers to 64-byte buffers. SPARROW understands the body of `get_string` that after the for-loop the argument buffer can be accessed (line 138) with index 64, an overrun.

- SPARROW understands integer arithmetic across procedure calls.

```
00    extern int signal;
01    extern int state;
02    int x = 0;
```

```
03
04    int work(int i) { return i % 11; }
05    int indexof()
06    {
07        int r = 0;
08
09        switch(signal) {
10            case 1:
11                r = 10 * x;  break;
12            case 2:
13                r = 20 * x + 2; break;
14            default:
15                r = 110*x + 3;
16        }
17        return work(r);
18    }
19
20    void setx()
21    {
22      if (state > 0 ) x = 1; else x = 2;
23    }
24
25    int foo()
26    {
27      int arr[10];
28
29      setx();
30      return arr[indexof()];
31    }
```

See the body of `foo`. Sparrow understands that the call to `setx` can set `x` to 1 or 2. Sparrow understands the subsequent call to `indexof` (line 30) can make `r` have 10 to 223, hence the return value `work(r)` (line 17) will be between 0 to 10. Sparrow thus concludes that the access `arr[indexof()]` (line 30) can overrun.

- Sparrow understands the loop-inducing state changes.

```
00    int foo()
01    {
02        int i;
03        int s = 1;
04        int arr[10];
05
06        for (i=0;i<10;i++)
07        {
08            arr[s] = 0;
09            s = s+1;
10        }
11        return 0;
12    }
```

See the for-loop. Sparrow understands that, inside the loop, variable `s` has 1 to 10, always larger than the loop-induction variable `i` by one. Hence Sparrow concludes that the access `arr[s]` overruns.

- Sparrow understands structure fields, pointers, and their inter-procedural effects.

```
00    struct Pair { char *x; char *y; };
01
02    struct Pair *make_pair(char *x, char *y)
03    {
```

```
04      struct Pair *p = (struct Pair *)malloc(sizeof(struct Pair));
05      p->x = x;
06      return p;
07    }
08
09    struct Pair *pair_this_big(int n)
10    {
11      char *p = (char *)malloc(n);
12      char *q = (char *)malloc(n);
13      return make_pair(p,q);
14    }
```

See the body of `pair_this_big`. SPARROW knows the pointers `p` and `q` point to two different memory blocks. SPARROW understands the call to `make_pair` will return a pointer to a structure whose one field points to what `p` points to, and that no field of the structure will contain the `q` pointer (line 5 and 6). SPARROW understands that the `p`-pointed memory block can be reached after the return (line 6) but the `q`-pointed one cannot. SPARROW concludes that the `q`-pointed memory block leaks at line 13.

- SPARROW understands loops, loop-escaping conditions, and global effects.

```
00    extern int g_num;
01
02    int loop()
03    {
04      int i;
05      int *p = (int *)malloc(sizeof(int)*100);
06
07      for (i=0; i<=10; i++) {
08        if (g_num < i) {
09          free(p);
10          return;
11        }
12      }
13    }
```

See the loop body. SPARROW knows that the `p`-pointed memory block can leak when `g_num` is larger than the loop bound 10. SPARROW knows that external variable `g_num` can be bigger than 10, hence SPARROW detects that the `p`-pointed memory may not be recycled.

- SPARROW understands inter-procedural effects on heap structures.

```
00    struct List{ int a; struct List * next; };
01
02    struct List *relink(struct List *x)
03    {
04      struct List *y, *t;
05      y = x->next;
06      free(x);
07      x = y;
08      while(x!=0){
09        t = x->next;
10        x->next = y;
11        y = x;
12        x = t;
13        }
14      t = (struct List *) malloc(sizeof(struct List*));
15      t->next = y;
16      return t;
17    }
18
```

```
19    struct List *foo()
20    {
21      struct List *node, *ret_val;
22      node = (struct List *) malloc(sizeof(struct List*));
23      node->next = NULL;
24      ret_val = relink(node);
25      return ret_val;
26    }
```

See the body of `relink`. Sparrow understands that the argument `x`'s first node is freed, its internal linked nodes are reversed (line 8-13), the reversed result is pointed to by a newly allocated node (line 15), and the newly allocated node pointer is returned. Sparrow concludes no leak for `relink`.

See the body of `foo`. Sparrow understands that the allocated `node` passed to `relink` will be freed inside `relink`. Sparrow hence concludes no leak for `foo` too.

- Sparrow understands malloc-and-free effects across recursive calls.

```
00    int *rec_free(int n)
01    {
02      int *y = (int *)malloc(sizeof(int)*10);
03
04      if(n>0){
05        free(y);
06        return rec_free(n-1);
07      } else {
08        free(y);
09        return (int *)malloc(sizeof(int)*10);
10      }
11    }
```

Sparrow understands that `ref_free` recursively calls itself, while it frees its allocated memory (line 2) either before its recursive call (line 5) or before its return (line 8). Sparrow hence detects no leak.

3.2 Supporting Various Dialects and Platforms

Sparrow can analyze your source written in almost all C and C++ dialects including:

ANSI C Arm CC GNU C/C++ Intel C/C++ TI C

Sparrow runs on various operating systems including:

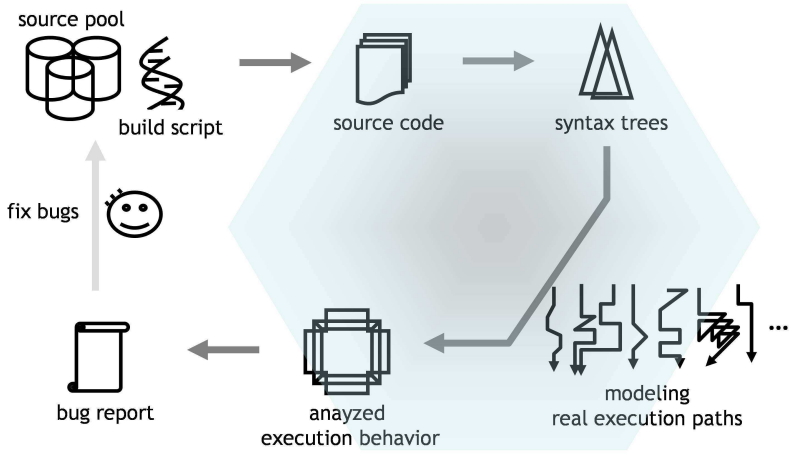
Cygwin IBM AIX FreeBSD Linux Mac OS X
Solaris Windows

4 How Sparrow Works

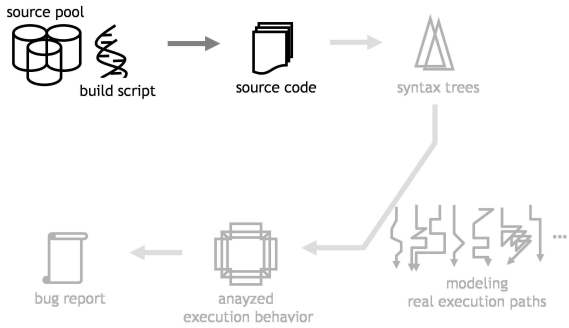
Sparrow is a one-button solution, from analyzing to bug-reporting. No change to your code or build scripts is necessary.

Sparrow's process consists of four all-automatic steps: understanding the code genetics, parsing and distilling the code, analyzing the code's run time behaviors, and reporting detected bugs.

Using Sparrow iterates. Sparrow reports bugs. The user verifies and fixes them, and runs Sparrow again to check the new revisions.



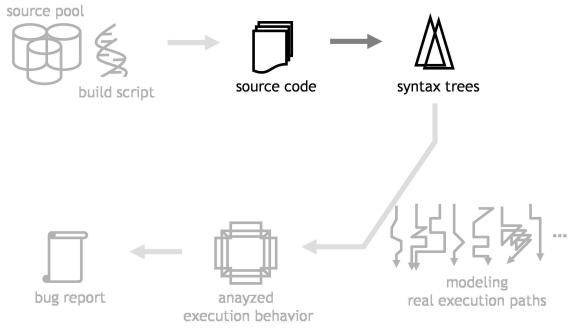
4.1 Step 0: Understanding the Code Genetics



Sparrow first identifies the source code that constitutes the target software. Sparrow infers from the build scripts of the target software which source files constitute the target and how their C/C++ macros and library files are preprocessed.

After this step, Sparrow has the complete set of vanila C/C++ code for the target, ready for the subsequent steps.

4.2 Step 1: Parsing and Distilling the Code

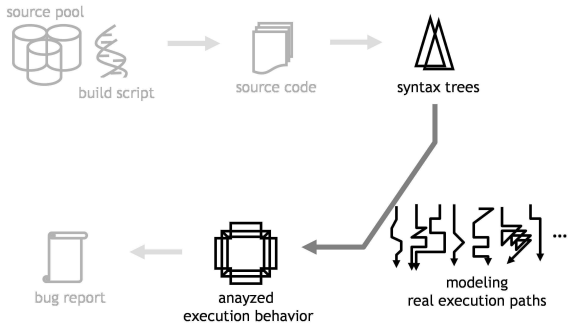


Sparrow parses the extracted source code into syntax trees. The syntax trees are two-dimensional structures that show how the target source is composed of

which language components in which order. Because SPARROW's parsing process is the same as in compilers, SPARROW has the complete understanding of the input source's syntactic structure.

SPARROW then distills the syntax trees to simplify the subsequent semantic analysis step.

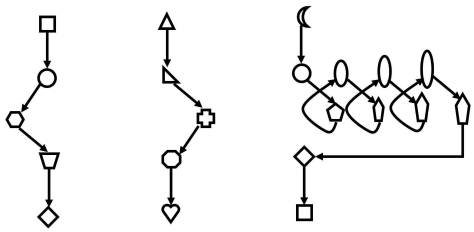
4.3 Step 2: Analyzing the Code's Run-Time Behavior



After understanding and distilling the syntactic structure of the code, SPARROW launches its semantic analysis phase. The semantic analysis is to analyze the input program's all possible execution scenarios.

This analysis phase can be explained in several ways. One way to explain it is rather mathematical. SPARROW captures the dynamics of the whole program executions in a set of equations. The unknowns of the equations are the program states during all possible executions. Because the exact solution of the equations (e.g., exact program states) cannot be computable or too costly, the equations must be approximately solved. From such approximate solutions (e.g., approximate program states) we find out where the program may have bugs.

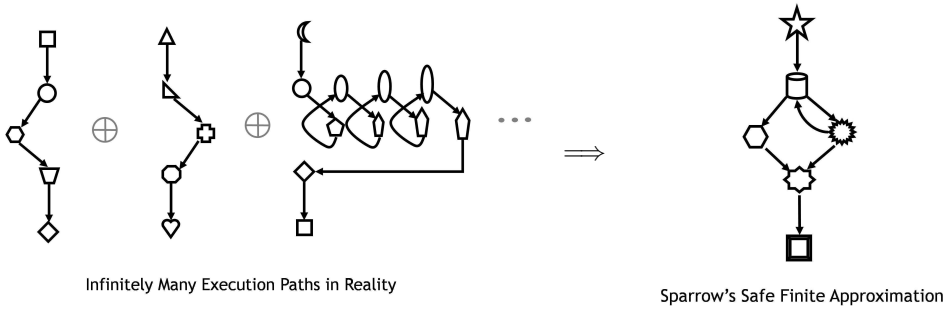
Another more intuitive explanation is also possible. SPARROW's analysis is a simluation of the input program's executions over a simplified, approximate space. Suppose the input program's real execution behavior includes the following three execution sequences, each of which corresponds to progam's three different inputs:



Three Execution Paths in Reality

Nodes in the execution sequence diagram represent the computer states at the corresponding program points. Edges represents the execution flow. The first sequence branches to the left, the second branches to the right, and the third iterates the right branch four times.

In reality, the program can have infinite or infinitely many execution sequences for varied input cases. SPARROW approximates such all possible execution sequences into a single or more, yet finite graphs:

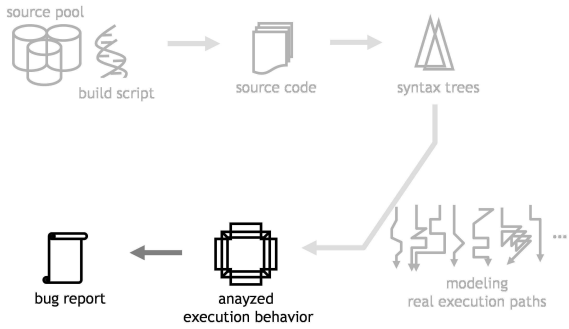


The multiple, exact program states at the same program point in different execution sequences is approximated to a single one.

This approximation is inevitable. Exact simulation covering all execution scenarios is too costly or simply impossible. The exact simulation cannot terminate if the program has an infinite loop, or needs an infinite number of scenarios if the program's input can be infinitely many.

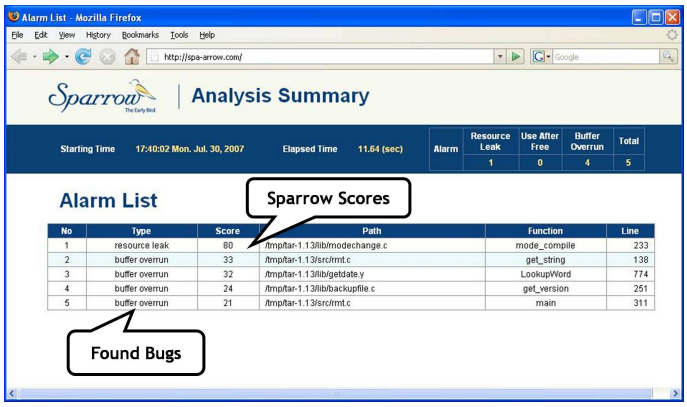
Though the approximation blurs the analysis, letting Sparrow sometimes spot non-bug places, such false alarms are minimized.

4.4 Step 3: Reporting Bugs



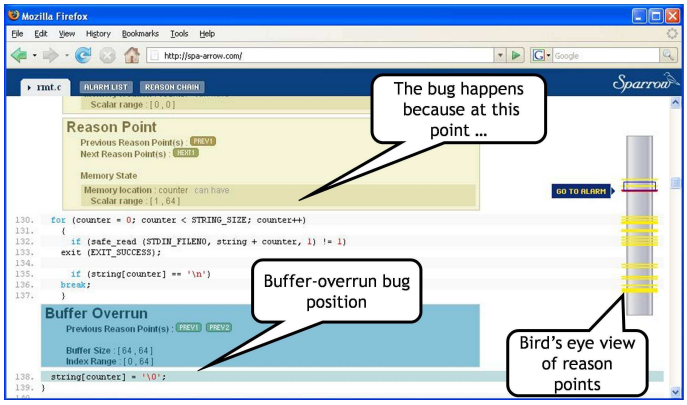
Having found bugs from analyzing the input program's approximate execution model, Sparrow reports the results to the user.

Sparrow uses statistical post-analysis. Given the reported alarms, classification methods compute a "strength" of each alarm being true. Sparrow uses the quantities, called "Sparrow scores" to rank the alarms, so that the user can check highly probable errors first. Sparrow uses the quantities also to sift out probable false alarms. Only the alarms that have trueness higher than a threshold are reported to the user. Sparrow's analysis result page lists discovered probable bugs with their Sparrow scores:



Sparrow's user interface is web-based. The interface is hyper-linked in-between source pages, helping the user to quickly traverse the source to verify the alarms.

Sparrow explains bugs. Because Sparrow in effect follows the execution flows of the source code, it can replay its analysis process back from the bug point. This replay is shown as hyper-linked "Reason Point" boxes overlayed in the source code. The chain of the reason points explains why a bug can happen. An example page of the source code with a bug point and its reason point chain is as follows:



5 Conclusion

5.1 Why Sparrow?

Your software size increases too fast. Conventional software quality assurance process fails to catch up.

You need an automatic tool suite to reduce the software quality assurance cost. A high-end, semantic-based static analysis technology is what is necessary in such tools.

Particularly, in the spectrum of the technology, you need one that finds most critical and relevant bugs with a reasonable price. You need a tool that sits in front of the second plateau of the technology spectrum curve. Sparrow stands there, right before the steep cost-increase point yet capable of finding most bugs in your code.

Sparrow's return on your investment is:

- **Early Detection**

  Sparrow enables you to fix bugs as early as possible – right when your source code is ready. If you find a bug of a large program at testing, it

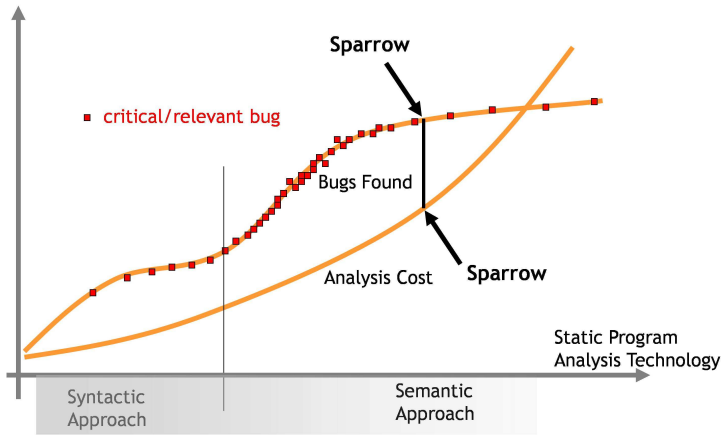


Figure 3: SPARROW's Position in the Spectrum of Static Analysis Technologies

is very costly to trace its cause back in the source code and even worse, if demands another round of expensive testing. SPARROW is a non-intrusive tool to your development environment. You can run it any time your source code is available.

- **Catch Deadly Bugs**

  SPARROW finds the most deadly bugs in C and C++ software. Buffer overruns and memory leaks are most common yet hard-to-find bugs that lead to serious malfunctions whose patterns are complicated and irregular. It is very expensive to locate such bugs simply by testing, failing to meet your tight time-to-market constraint.

- **Cost Reduction**

  SPARROW catches bugs early in the development cycle. Without extra adjustments or implentation for testing environment, you can save quite significant amount of testing time and resources with SPARROW. Early detection of bugs – and the most deadly ones – is what SPARROW guarantees, providing innovative cost reduction benefits for software debugging and testing.

5.2 Deploying SPARROW

SPARROW is used in two ways:

- As a tool in daily build process
- As a tool in periodic code-review process

In the first case, SPARROW is integrated with the machine that does the nightly build. SPARROW scans new versions of code every time it is compiled. The ouput is reviewed by the developers to quickly fix any unnoticed error.

In the second case, SPARROW scans software source before the software is released or shipped, usually before regular pre-release tests. The output is reviewed by the developers or independent verification and validation test teams. SPARROW is also used to quality-check source code provided by subcontractors.

5.3 Free Trial

We offer a free trial of SPARROW and comparison benchmark tests. To find out more, contact us at `sales@spa-arrow.com`.

SPARROW homepage:	`http://www.spa-arrow.com`
Inquiry:	`sales@spa-arrow.com`

□