ISSN 0361-7688, Programming and Computer Software, 2010, Vol. 36, No. 4, pp. 225–236. © Pleiades Publishing, Ltd., 2010.
Original Russian Text © I.K. Isaev, D.V. Sidorov, 2010, published in Programmirovanie, 2010, Vol. 36, No. 4.

The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs

I. K. Isaev and D. V. Sidorov

Institute for System Programming, Russian Academy of Sciences,
ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: iisaev@ispras.ru, sidorov@ispras.ru

Received December 28, 2009

Abstract—The article presents Avalanche—a dynamic analysis defect detection tool. Avalanche uses dynamic instrumentation provided by Valgrind [1] to collect and analyze the trace of program execution. The result of such an analysis is a set of input data which either shows an error in the program or allows next iteration to cover fragments of the program that were not yet executed and, therefore, checked. Thus, starting from a single test case, Avalanche implements iterative dynamic analysis, repeatedly executing the program with various automatically generated test data, while each execution increases the coverage of code. The article describes Avalanche internals, and discusses the results of analysis of several open source projects with Avalanche, which resulted in detection of over 15 bugs. Many of the detected bugs are confirmed and fixed by developers.

DOI: 10.1134/S0361768810040055

1. INTRODUCTION

Dynamic program analysis was long considered to be a too heavyweight approach to defect detection; its results didn't justify the effort and resources required. However, there are two important trends in the modern software development industry which allow a new look at this problem. On the one hand, as the size and complexity of software steadily increase, any automatic defect detection tool can prove to be helpful. On the other hand, the steady increase in the performance of modern computer systems makes it possible to solve more and more complicated computational problems efficiently. The recent surveys in the field of dynamic analysis, such as SAGE [2], EXE [3], and Flayer [4], show that, despite the complexity inherent in the dynamic analysis approach, it can be successfully applied for at least a certain kind of programs. In addition to these research projects, we should note Valgrind [1]—one of the most successful open source projects in this field. Its popularity on Linux platform proves that the use of dynamic analysis is justified in the case if it is able to reliably detect actual bugs and can work completely automatically, that is, without any interference of programmer or tester.

Static analysis is often considered to be a more efficient alternative to dynamic analysis. Following this approach, defect detection is carried out without running the program but rather by analyzing the program source code. Typically, an abstract model of the program is constructed, which is an actual object of the analysis. We would like to highlight the following key features of static analysis:

• Separate analysis of different program pieces (typically, functions or procedures) is possible, which provides a rather efficient way of dealing with the nonlinear growth of the complexity of the analysis.

• False positives are possible. They may occur due to either the loss of some details during the model construction or incomplete analysis of the model.

• There are two problems when a bug is reported. First, it should be checked whether it is true or false positive. Second, the input data for reproduction of the bug sometimes need to be found.

Unlike the static analysis, the dynamic analysis is performed during the program execution. In this case

• The program needs some input data to run

• The dynamic analysis can detect bugs only on the trace determined by the provided input data; the bugs in the other parts of the program are not going to be detected.

• In most implementations false positives are impossible because a bug is detected at the moment of its occurence; therefore, the detected bug is not a prediction made as a result of analysis of the program model but rather a statement of the fact.

Thus, the problem of input data selection is rather important for defect detection. If static analysis tools are used, the input data for detected bugs should be found in order to check their true positiveness and locate the best place for the fix. In the case of dynamic analysis, the input data is chosen from the considerations of either the maximum code coverage or the possibility of analysis of the most interesting places.

So, the successful choice of the input data is crucial for the efficiency of the dynamic analysis.

Various methods that make it possible to use dynamic analysis both for defect detection and for generation of input data reproducing the defect have been proposed recently. The essence of these methods may be described as follows. They introduce the notion of symbolic or tainted data—any data that program obtains from an external source (standard input stream, files, environment variables, etc.). All the information about the uses of tainted data in the program is somehow collected; then this information is written in the form of boolean constraints on the tainted data values (this constraints may include information about all conditional jumps depending on the tainted data and information about the usages of tainted data in potentially dangerous places of the program). If the values satisfying the above constraints may be calculated, this may indicate either the possibility of an error occurrence or the ability to cover other parts of the program than those that were covered in the previous runs of the program.

In this paper, we describe Avalanche—the tool that implements such an approach on the basis of the open source dynamic binary instrumentation framework Valgrind [1] and the constraint checking solver STP [5]. The following features were our main targets:

• the analysis should be as complete as possible;

• the bugs should be detected effectively and without any false positives;

• the input data which reproduces detected bugs should be generated.

At the initial stage of the development, it was decided to restrict the probable set of analyzed programs; Avalanche may analyze only the programs that read input data from a single input file. Any data received from other sources (other files, network sockets, environment variables) is ignored. Below we assume that the program under analysis deals with a single input file.

The paper is organized as follows. In Section 2 we briefly present the general scheme of the tool. Sections 3–5 give more detailed description of the components of the tool (Section 3 describes the tracking of the tainted data flow in the program, the modeling of Valgrind intermediate representation with STP declarations, the generation of constraints for the traversal of new parts of program, dangerous operations of the Valgrind intermediate representation and generation of constraints for their verification. Section 4 introduces the strategy for the traversal of the conditional jumps tree of the program.) In Section 6 we discuss the results of applying Avalanche to ten open source projects; we provide the detailed statistics about the results of the analysis and list detected bugs.

2. GENERAL WORKFLOW SCHEME.

Avalanche consists of four main components: two Valgrind plugins—Tracegrind and Covgrind, constraint checking solver STP and the driver module. Tracegrind dynamically tracks the flow of tainted data in the program and collects conditions for traversing previously untraversed parts of the program and for the execution of dangerous operations. The driver passes these conditions to STP to check their satisfiability. If some condition is satisfiable, then STP determines the values of all the variables involved (including the values of the bytes of the input file) that turn those conditions true.

• If some of the conditions for checking the execution of dangerous operations is satisfiable, the driver runs the program once more (this time, without any instrumentation) with the corresponding input file in order to confirm the detected bug.

• The satisfiable conditions for the traversal of untraversed parts of the program determine a set of input files for new runs of the program. Thus, after each run, STP automatically generates a set of input files for the subsequent runs of the analyzer. Therefore, there is a problem of selecting the most interesting input data from this set. The data, which is more likely to trigger a bug should be hanled first. To solve this problem Avalanche uses a heuristic metric—the number of previously untraversed basic blocks (the notion of basic block is the same as in Valgrind framework [1]). For calculation of the heuristic value Covgrind plugin is used, which also registers the occurrence of possible run-time errors. Covgrind is a much more lightweight plugin than Tracegrind; therefore, it is possible to calculate the heuristics for all the candidate input files relatively fast and select an input file with its greatest value. Below, we consider each of the listed components in more details.

3. TRACEGRIND: TRACKING THE FLOW OF TAINTED DATA

The main tasks solved by this component are as follows:

• tracking the flow of tainted data in program;

• composition of conditions that define

1. the possibility to execute (or not to execute) a conditional jump in the program;

2. the possibility to execute a potentially dangerous operation.

Whether or not an operation is considered to be potentially dangerous, depends on the types of defects that need to be detected. For instance, if null pointer dereferencies are to be detected, then each memory access operation (read or write) should be considered as dangerous. Currently, Avalanche keeps track of the following operations

• all divisions (possible division by zero);

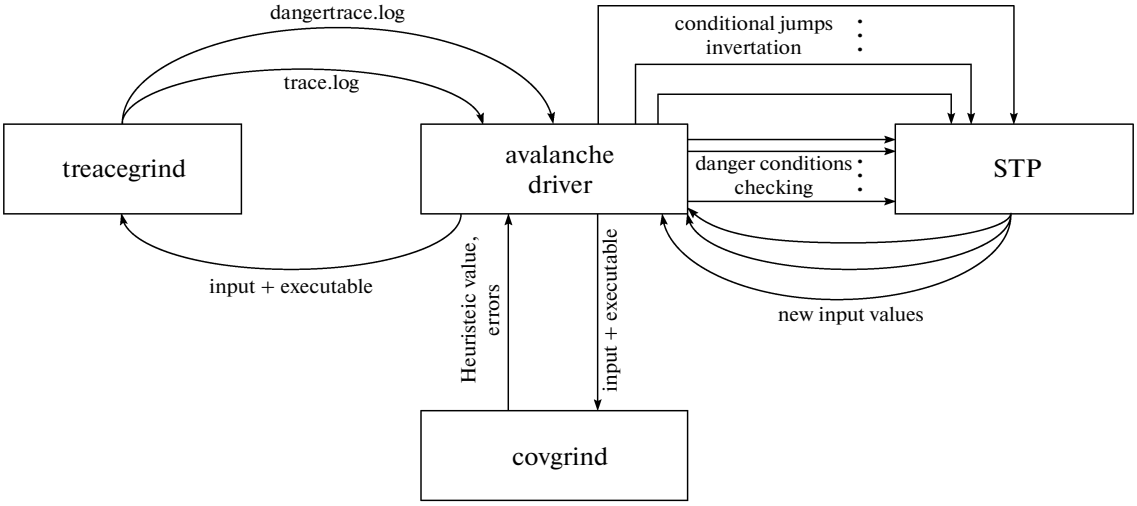

Fig. 1. Avalanche: General Scheme.

• all memory read and write operations (possible null pointer dereference).

In both cases, the use of a zero value as operand of these operations causes termination of the program (the former results in a segmentation fault, the latter—in a floating point exception). Therefore, the conditions composed by Tracegrind should reflect the fact of a divisor or memory access address being equal to zero.

Let us discuss the ways to solve these tasks.

3.1 The Flow of Tainted Data

To track the sources of tainted data, Tracegrind uses the feature of Valgrind, which allows interception of system calls in the instrumented program. In particular, the following file handling system calls are intercepted:

- `open`,
- `close`,
- `lseek`,
- `read`,
- `map`.

The name of the file which the program reads the data from must be passed over to Avalanche as an option (later, driver passes this option to Tracegrind each time it is executed). Tracegrind keeps track of the interaction of the program with only this file, and all the data read from it is marked as tainted. The data that is read from other files is not marked as tainted. Currently, Tracegrind supports only single file as a source of tainted data; later we hope to overcome this restriction.

The flow of tainted data is tracked by explicitly checking the operands of every instruction in the Valgrind intermediate representation. If any of the operands is tainted, then the result of the operation is marked as tainted as well. Initially, all the memory locations which are filled with data during the execution of `read` and `map` system calls are marked as tainted. Then, if any of these values is read into a temporary variable, this variable is marked as tainted as well. The use of tainted variables as operands of various operations (arithmetic, bitwise, and so on) also causes the resulting temporary variable to become tainted. Memory write and register read and write operations are handled in the same way. If a tainted value is rewritten (e.g., a constant or the value of an untainted variable is stored to a register), the corresponding object (memory location or register; a temporary variable can be written only once) is not marked as tainted any longer.

Such an approach to tracking the flow of tainted data in a program is simple and rather efficient. However, a program can contain implicitly tainted values, that is, values that does depend on the contents of the input file but which has no explicit assignments that make it possible to track their dependence on the data read from the input file.

Example:

```
int len, fd;
...
read(fd, &len; sizeof(int));
int i = 0;
for (int j = 0; j < len; j++) {
  i++;
}
```

In this example, the value of `i` is definitely determined by the contents of the file; however, it is impossible to establish this fact using the approach described above; the value of `i` is (erroneously) assumed to be untainted. Another example of implicitly tainted value is the size of the input file.

```
memory_1 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_0
           WITH [0hex04057000] := file_input_dot_txt[0hex00000000];
```

Fig. 2.

3.2 Composition of STP Declarations

To compose conditions in order to later check them by the means of STP, every instruction of the Valgrind intermediate representation that operates on tainted data is translated into an input declaration for STP. Let us consider the principles of this translation. The translation is performed "on the fly" at run time by the instrumentation code of Tracegrind. The instrumented program produces two traces. Each trace is a text file containing a sequence of declarations for the STP. The contents of the traces have much in common; the only difference is that the first trace contains the conditions for checking the possibility of traversing previously untraversed parts of the program (below this trace is referred as "the conditional jumps trace" and is referred by trace.log); the second trace includes conditions for checking possible errors while executing dangerous division and memory access operations (below this trace is referred as "the trace with dangerous operations" and is denoted by dangertrace.log). The driver parses these traces and emits separate queries to the STP. Consider the creation of declarations in more details (below, we assume, if no otherwise specified, that the declarations fall into both traces).

3.2.1. Modeling of Entities. The input file, the set of registers, and the address space (memory) of the program are represented by arrays in STP. The elements of all three arrays are eight-bit vectors. The arrays representing the address space and the input file are indexed by 32-bit indices, and the array of registers is indexed by eight-bit indices. This is how declarations of a memory array and a register array look:

```
memory_0 : ARRAY BITVECTOR(32) OF BITVECTOR(8);
register_0 : ARRAY BITVECTOR(8) OF BITVECTOR(8);
```

The array representing the input file is declared similarly; its name includes the name of the input file (input.txt):

```
file_input_dot_txt : ARRAY BITVECTOR(32)
OF BITVECTOR(8);
```

The temporary variables are represented by bit vectors of corresponding length (1, 8, 16, 32, or 64). The name of each variable (according to the STP syntax, all the declared objects must have unique names) includes the address of the basic block where the variable belongs, the number of the variable within this block, and the number of the current execution of the block:

```
t_409f9d8_59_2 : BITVECTOR(8);
```

This declaration corresponds to the one-byte variable with number 59 in the basic block with the initial address `409f9d8`, besides, the declaration was created during the third execution of the basic block (this is indicated by the last digit 2 because executions are enumerated starting from zero).

3.2.2. Declarations for System Calls. When the `read` and `map` system calls are intercepted, a bunch of declarations is created; these declarations represent the equality of the values read from input file to the values in those memory cells, where the input data was written (see Fig. 2).

This declaration models the reading of the first byte from the file `input.txt` into the memory cell with the address `0x04057000`. Generally, every use of `read` and `map` results in the appearance of a chain of such declarations. The number of such declarations is equal to the number of the bytes read.

3.2.3. Declarations for Instructions Operating on Tainted Data. All instructions are also modeled in a quite an obvious way. For example, the memory read instruction `t17 = LDLE:I8(t15)` (the one byte value from the cell with the address contained in the `t15` variable is loaded into `t17` variable) produces the declaration

```
ASSERT(t_401ff8_17_0=memory_712[0hex04057000]);
```

If the variable `t15` is tainted, the declaration representing the condition of `t15` being equal to zero is added to the trace with dangerous operations. Furthermore, the syntactic directive `QUERY(FALSE)` which is a hint for STP to check the satisfiability of the formula is also added there:

```
ASSERT(t_401ff8_15_0=0hex00000000);
QUERY(FALSE);
```

Translation of instructions that read or write more than one byte at once is a bit more complicated. Since the elements in the modeling arrays are only one byte long, the value of a long variable has to be either decomposed into separate bytes (using the selection operation) or assembled from individual bytes using

```
memory_717 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_716
             WITH [0hexbec91560] := t_40139a0_0_22[7:0];
memory_718 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_717
             WITH [0hexbec91561] := t_40139a0_0_22[15:8];
memory_719 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_718
             WITH [0hexbec91562] := t_40139a0_0_22[23:16];
memory_720 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_719
             WITH [0hexbec91563] := t_40139a0_0_22[31:24];
```

Fig. 3.

concatenation and bitwise disjunction. Assume that the variable `t0` in the instruction `STle(t22) = t0` is tainted and is four bytes long. Then, the declarations shown in Fig. 3 will be created.

Here, the lower bytes of the variable `t22` are written to the bytes at lower addresses. This is modelled by using the selection of the corresponding bits. In the case of reading, the inverse operation is performed; a long value is assembled from the pieces. For example, the instruction `t2 = GET:I32(0)` (suppose register 0 contains a tainted value) produces the declaration shown in Fig.4.

Register `eax` is associated with register 0 in the Valgrind representation; register `al` is also associated with zero (as the lower part of `eax`), register `ah` is associated with one, etc. Each of the four bytes is obtained from the array; then, the vectors are concatenated with null vectors of the corresponding length and joined using the bitwise disjunction.

Arithmetic, bitwise and type conversion operations in the Valgrind intermediate representation are modeled by the corresponding STP operations. For comparison operations, the `IF THEN ELSE` operation is also used. For example, the operation `t10 = CmpEQ32(t11, 0x0:I32)` produces the declaration shown in Fig. 5.

The addition operation `t11 = Add32(t2m 0x1:I32)` generates the declaration shown in Fig. 6.

The signed type convertion operation is modeled by the analogous STP operation. For example, the operation `t16 = 8Sto32(t17)` corresponds to the declaration

```
ASSERT(t_40d1ff8_16_0=BVS0X(t_40d1ff8_17_0, 32));
```

If the second operand of the division operation is tainted, the declaration representing the condition of divisor being equal to zero is added to the trace with dangerous operations; furthermore, the directive `QUERY(FALSE)` is also added. For example, the operation `t18 = DivModU64to32(0x3B9AC9F4:I64, t16)`, produces the following declaration in the trace with dangerous operations:

```
ASSERT(t_4053000_16_0=0hex00000000);
QUERY(FALSE);
```

At the current stage of development, only integer operations are instrumented and cause the generation of STP declarations. Floating point operations apparently are not going to be supported as they are hard to model by the available STP operations.

3.2.4. Conditional Jumps. If the conditional jump instruction is executed and the condition depends on the tainted variable, then, depending on the outcome of the jump, the declaration saying that the variable is true or false is produced. Besides, the directive `QUERY(FALSE);` is added to the conditional jumps trace. For example, the instruction of not taken conditional jump `if (t3) goto 0x40D3CC1:I32` produces the declarations

```
ASSERT(t_40d3cab_3_0=0bin0);
QUERY(FALSE);
```

```
ASSERT(t_40d3cab_2_0=((0hex000000 @ registers_4[0hex00]) |
                      (0hex0000 @ registers_4[0hex01] @ 0hex00) |
                      (0hex00 @ registers_4[0hex02]) @ 0hex0000) |
                      (registers_4[0hex03]) @ 0hex000000)));
```

Fig. 4.

```
ASSERT(t_40d3cab_10_0=IF t_40d3cab_11_0=0hex00000000
                      THEN 0bin1 ELSE 0bin0 ENDIF);
```

Fig. 5.

```
ASSERT(t_40d3cab_0_0=BVPLUS(32,t_40d3ca b_2_0,0hex00000001));
```

Fig. 6.

4. DRIVER MODULE

The driver integrates all the other Avalanche components together. Its main functions are as follows:

• coordination of the interaction between the processes of the other components;

• control of the traversal of the conditional jumps tree of the program

After the Tracegrind module has finished, the driver parses the two traces produced by Tracegrind. Schematically, their contents look approximately as follows:

trace.log dangertrace.log

trace.log	dangertrace.log
ASSERT(t1=1);	ASSERT(t1=1);
QUERY(FALSE);	ASSERT(t2=1);
ASSERT(t2=1);	ASSERT(t_div=0);
QUERY(FALSE);	QUERY(FALSE);
ASSERT(t3=0);	ASSERT(t3=0);
QUERY(FALSE);	ASSERT(t_deref=0);
…	QUERY(FALSE);
ASSERT(tn=0);	…
QUERY(RALSE);	ASSERT(tn=0);

Here, ASSERT(ti=1(0)) denotes the conditions corresponding to the outcome of conditional jumps that depend on the tainted data; ASSERT(t_div=0) (ASSERT(t_deref=0)) denote the conditions of divisor (memory access address) being equal to zero. The declarations generated by other instructions are omitted for the sake of clarity and because of space limitations.

First, the driver parses dangertrace.log and constructs the following queries according to its contents:

```
ASSERT(t1=1);      ASSERT(t1=1);
ASSERT(t2=1);      ASSERT(t2=1);
ASSERT(t_div=0);   ASSERT(t3=1);
QUERY(FALSE);      ASSERT(t_deref=0);
                   QUERY(FALSE);
```

Note that the satisfiability of these queries means the occurrence of errors in the dangerous operations on the same trace of the program. If STP decides that the queries are satisfiable, then, given the values of the variables that satisfy them, the driver constructs the new contents of the input file and runs the program with it to check the actual occurrence of error. If the error is confirmed, the exploit input file is saved by the driver.

Next, the control module parses trace.log and passes the following queries to STP in order to check their satisfiability (see Fig. 7).

Note that in the ith query the first $i - 1$ conditions of the conditional jumps remain direct (i.e., they have the same outcome as during the execution of the program), and the ith condition is inverted (i.e., if the jump was not taken, then ASSERT(ti=1) is added to the query; otherwise, the condition ASSERT(ti=0) is added). If the request is satisfiable, the driver saves the contents of a new input file which enables the execution of a new trace of the program. After checking the satisfiability of all the requests, the driver has a set of new input files.

Figure 8 shows the conditional jumps tree of the program. The conditional jumps are depicted by vertices, the lower outgoing edge denotes the direct execution of the program, the upper outgoing edge denotes the inverted execution (i.e., the execution defined by the data obtained as a result of the successful satisfiability checking of the corresponding query to the STP). The very first execution of the program is associated with the path 1–2–4–8–16.... Suppose that after the first execution all the queries to invert the outcome of the conditional jumps turned out to be successful. Then, at the next step, the driver has the values of the input data needed to traverse the branches 1–3–6–12–24..., 1–2–5–10–20..., 1–2–4–9–18..., 1–2–4–8–17..., and so on. For that reason, the first execution is depicted by a dashed line that cuts off the lowest branch of the tree. As it has already been mentioned, from the set of all possible inputs the driver selects the file which achieves the greatest increase of coverage measured in previously uncovered basic blocks of the analyzed program. To count the number of such blocks, the driver runs the analyzed program with Covgrind plugin and each of the available input files.

Suppose that the branch 1–3–6–12–24... gives the greatest increase of coverage. Then, after the second run of the program with Tracegrind, the driver acquires new input files that cover the branches 1–3–

```
ASSERT(t1=1); ASSERT(t1=1); ASSERT(t1=1);       ASSERT(t1=1);
QUERY(FALSE); ASSERT(t2=0); ASSERT(t2=1);       ASSERT(t2=1);
              QUERY(FALSE); ASSERT(t3=1);   …   ASSERT(t3=0);
                            QUERY(FALSE);       …
                                                ASSERT(tn=1);
                                                QUERY(FALSE);
```

Fig. 7.

7–14–28..., 1–3–6–13–26..., and 1–3–6–12–25... (in addition to the files generated after the first run and not yet used for the program execution). Among them, the file providing the best coverage is again selected, and the program is run with Tracegrind once more, and so on. Thus, the branch-by-branch traversal of the conditional jumps tree is performed, and the files that provide the coverage of previously untraversed branches are collected. Obviously, the complete traversal of the tree requires an exponential number of runs (2^{n-2}, where n is the depth of the tree or, in other words, the number of conditional jumps that depend on the tainted data).

Besides, each node requires to solve the NP-complete problem to check the satisfiability of the corresponding boolean formula. Therefore, even processing of a single branch can be very time consuming. To speedup the traversal of the tree, the threshold value of lookup depth may be specified as one of the Avalanche options; then, not more than the specified number of conditional jumps will be inverted on each branch, while the jumps laying deeper than the specified threshold will generate no queries to the STP (this behavior corresponds to cutting off the lower parts of the tree).

5. COVGRIND: MEASURING THE COVERAGE ACHIEVED BY TEST CASES

The component Covgrind is extremely simple. It dumps a list of addresses of the executed basic blocks. The driver uses this list to calculate the number of previously uncovered blocks. Furthermore, at the startup Covgrind sets up a timer. If the timer fires while the program is executed (that is, the specified time has elapsed), Avalanche interprets it as the presence of an infinite loop in the program. The timer value is an Avalanche starting option. If Covgrind terminates abnormally (exit on a signal), it is assumed that an error in the program occurred. In both cases (infinite loop and abnormal termination), the driver saves the input file used to execute the program.

6. RESULTS OF ANALYSIS OF REAL-LIFE APPLICATIONS

6.1. Projects Analyzed

The effectiveness of Avalanche in detecting bugs was explored using a large number of open source projects. In this paper, we examine the results obtained for the following ten projects.

qtdump (libquicktime-1.1.2). Libquicktime is a library for reading and writing quicktime/avi/mp4 files. Qtdump is a wrapper for the library function quicktime_dump(file) that prints the parsed contents of the input file.

flvdumper (gnash-0.8.6). Gnash is a GNU flash player. The program flvdumper prints the internal information contained within an flv videofile.

cjpeg (libjpeg7). Libjpeg is a library for handling JPEG files. Cjpeg converts bmp files into jpeg.

sndfile-mix-to-mono (sndfile-tools-1.02). Libsnd- 2 2
file is a library for reading and writing MS Windows
WAV and Apple/SGI AIFF files. Sndfile-tools is a set 2
of utilities using libsndfile. Sndfile-mix-to-mono con- 2
verts a multi-channel input file to a mono output file,
by mixing all input channels into one.

swfdump (swftools-0.9.0). SWFTools is a collection of utilities for working with Adobe Flash files (SWF files). Swfdump prints out various informations about SWF files, like contained images/fonts/sounds, disassembly of contained code, cross-reference and so on.

avibench (avifile). Avifile is a player and library for working with multimedia AVI files. Avibench is used to collect benchmarks for the functions of the avifile library.

xmllint (libxml2-2.7.6). Libxml2 is a library for parsing xml-files. Xmllint is a parser based on libxml2.

mpeg2dec (libmpeg2-0.5.1). Libmpeg2 is a library for decoding mpeg-2 and mpeg-1 video streams. Mpeg2dec is a simple test utility for libmpeg2, it decodes mpeg-1 and mpeg-2.

mpeg3dump (libmpeg3-1.8). Libmpeg3 is a library for decoding compressed MPEG files. Mpeg3dump is an utility used for dumping data into a 24-bit PCM file or extracting audio from it.

speexenc (speex-1.2rc1). Speex is an open source format designed for speech. Speexenc compresses WAV and uncompressed files into the speex format.

Table 1. Statistics for the analysis with initially incorrect input data

2	qtdump	flv-dumper	cjpeg	sndfile-mix-to mono	swf-dump	avi-bench	xmllint	mpeg2-dec	mpeg3-dump	speex-enc
total number of runs	126	97	112	10	355	37	60	147	34	5
initial coverage	2927	5853	2259	2395	2202	6037	3918	2503	2757	3507
increase of coverage	1667 (57%)	16 (<1%)	1188 (53%)	821 (34%)	2010 (91%)	1357 (22%)	2231 (57%)	206 (8%)	1885 (68%)	32 (1%)
prediction accuracy	95%	100%	96%	100%	70%	86%	100%	100%	93%	100%
Tracegrind time	4%	84%	3%	<1%	10%	3%	2%	4%	5%	1%
STP time	61%	4%	96%	99%	54%	80%	93%	50%	73%	<1%
Covgrind time	34%	11%	1%	<1%	35%	17%	4%	46%	21%	98%
dangerous operations	173	0	404	0	159	1576	81	0	585	0
number of defects	5	0	1	1	2	1	0	0	2	0
test cases	82	0	35	1	437	2	0	0	3136	24
t_{min}	491	–	252	8	55	3041	–	–	5	–
t_{max}	5827	–	252	8	684	3041	–	–	5	–

Table 2. Statistics for the analysis with partially correct input data

2	qtdump	flv-dumper	cjpeg	sndfile-mix-to mono	swf-dump	avi-bench	xmllint	mpeg2-dec	mpeg3-dump	speex-enc
total number of runs	3	83	20	1	293	28	58	145	62	3
initial coverage	3060	6994	3098	2497	2770	6077	4617	5131	3195	3537
increase of coverage	103 (4%)	569 (8%)	311 (10%)	0	2350 (85%)	1135 (19%)	1953 (42%)	25 (<1%)	1457 (46%)	54 (2%)
prediction accuracy	100%	89%	95%	–	68%	86%	100%	100%	94%	100%
Tracegrind time	<1%	58%	<1%	<1%	9%	3%	2%	5%	5%	<1%
STP time	1%	28%	99%	>99%	61%	80%	93%	73%	75%	3%
Covgrind time	99%	10%	<1%	<1%	29%	17%	4%	21%	20%	97%
dangerous operations	0	663	214	1	231	3075	104	0	585	0
number of defects	1	1	1	0	2	1	0	1	2	0
test cases	24	2	37	0	933	2	0	127	3136	25
t_{min}	314	156	3	–	59	2919	–	12	1	–
t_{max}	314	156	3	–	248	2919	–	12	9	–

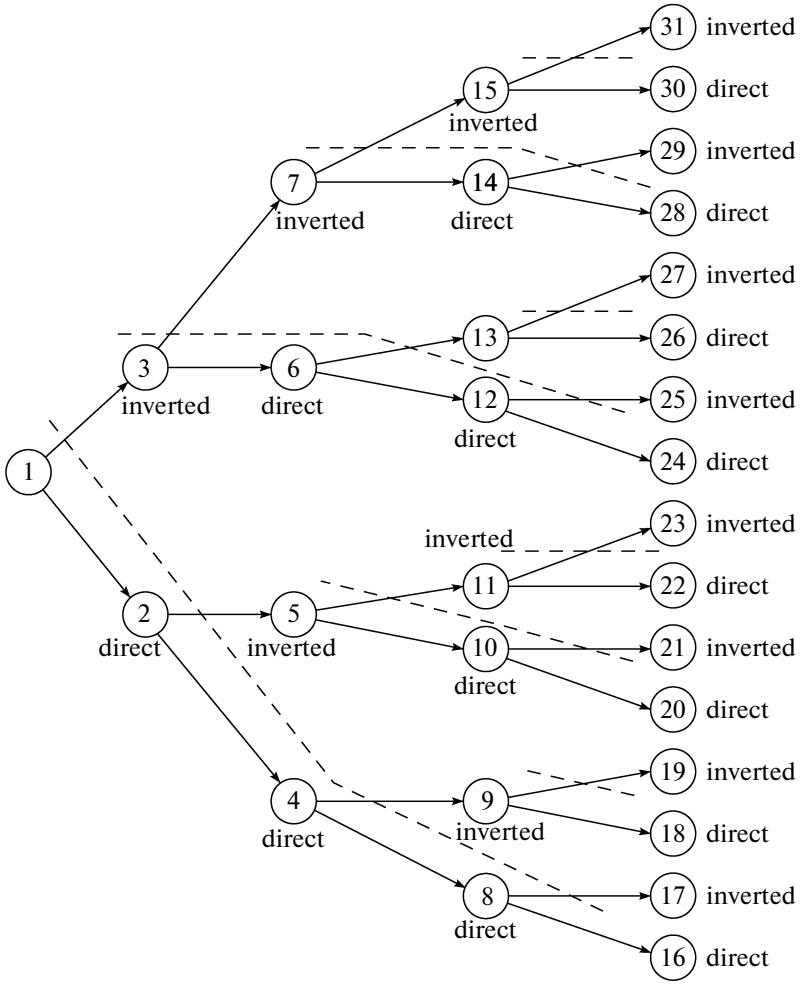

Fig. 8. Conditional jumps tree of the program.

6.2. Analysis Parameters and the Detailed Results

The programs were run using two types of input files. The files of the first type were short (712 bytes) and did not match the format expected by the analyzed programs; i.e. they were malformed. On the contrary, the files of the second type were formed correctly (i.e. they matched the expected format), but they were cut to make them 712 bytes long. The size of the input files was limited in order to reduce the number of constraints produced and speedup the checking of their satisfiability. The option for the lookup depth was set to 100 conditions (depending on the tainted data). The option for the timeout value (for detection of infinite loops) was set to 300 s. Avalanche worked for 7500 seconds on each of the above projects, then it was forced to terminate (i.e. the conditional jumps tree of the program was never traversed completely).

The results are presented in Tables 1 and 2.

The *total number of runs* is the number of program runs with the Tracegrind plugin (this number is also equal to the number of partially traversed branches of the conditional jumps tree).

Prediction accuracy is the percentage of runs when the execution followed exactly the branch predicted according to the satisfiability of the constraint obtained by inverting the outcome of one of the conditional jumps (as compared with the preceding run of the program). Let us make some conjectures that could explain why the prediction accuracy does not reach 100%.

• Approximations made during the translation of Valgrind intermediate representation into STP declarations. For instance, the syntax of STP declarations requires the second operand of the binary shift operation to be specified as an explicit number. If during the execution of the program the second operand turns out to be a tainted variable, the value of this variable has to be specified, which can cause an inexact result when the satisfiability of the resulting formula is checked.

• Nondeterminism of program execution. For example, if the program uses arithmetic operations dealing with addresses located in dynamic memory, it

can can result in different traces even if the program is run on the same input data.

• The existence of implicitly tainted data.

Initial coverage is the number of basic blocks executed at the first run of the program.

Increase of coverage is the total number of basic blocks executed during the analysis done by by Avalanche (i.e. it is the number of basic blocks that were not executed during the first run of the program but were executed during later runs). The coverage includes not only the basic blocks of the program itself, but also the basic blocks of the libraries used in the program. The ratio of the increase of coverage to the initial coverage is also given measured in percents.

Tracegrind, STP, and Covgrind time are the ratios of the execution time of each of these components to the total execution time measured in percents.

Dangerous operations is the overall number of formulas checked to find bugs in dangerous operations.

Test cases is the total number of exploits generated by Avalanche during the analysis (input files that cause exit on signal or an infinite loop).

Number of defects is the number of unique defects reproduced using the test cases generated by Avalanche. The reason for the fact that the number of test cases exceeds the number of unique defects is that the same bug may happen on different traces.

t_{min} is the time (in seconds) elapsed from the start of Avalanche until the first bug was found.

t_{max} is the time (in seconds) elapsed from the start of Avalanche until the last unique bug was found.

For qtdump, the analysis with initially malformed input data proved to be more effective than the analysis with partially correct data—the former resulted in detection of five defects, while the latter found only one bug, which was already included in the list of bugs detected in the first run of analysis. The situation for
2 sndfile-mix-to-mono is similar: when the analysis began with partially correct data, the checking of satisfiability of the generated conditions took too long time; as a result, the coverage didn't increase in the assigned amount of time and, therefore, no bugs were
2 detected. On the contrary, the analysis of sndfile-mix-to-mono started with incorrect input data caused a detection of a bug.

For flvdumper and mpeg2dec, the situation is completely reverse. The analysis started with initially incorrect input data did not detect any bugs, while the analysis started with partially correct data found certain defects. In general, the initial execution of the analyzed application with the expected input data immediately increases the part of the tree of conditional jumps that need to be explored further. This can result either in a faster detection of bugs or in a slowdown of the analysis due to the expansion of size and amount of the constraints that need to be checked.

For swfdump, avibench, xmllint, mpeg3dump, and speexenc the two runs showed no significant difference. In cjpeg a bug was found much faster when the analysis started with partially correct data.

Note that a relatively large proportion of the analysis time is taken by Covgrind. This can be due to the fact that in order to measure the metric Covgrind is executed much more often than Tracegrind. The other factor is the use of the timer for detecting infinite loops. If the program may enter the same infinite loop on different traces, it may cause a significant waste of time because when executing each of these traces Covgrind awaits for the expiration of the assigned timer.

6.3. Brief Review of Detected Bugs

qtdump (libquicktime-1.1.2). Three bugs are null pointer dereferences, another one is an infinite loop, and, the last one causes a segmentation fault. Some of these bugs were fixed by the developer.

flvdumper (gnash-0.8.6). The direct cause of the bug is the occurrence of exception in the boost library used in the program (one of the internal boost pointers turns out to be null). Since the program does not handle this exception, the program receives SIGABRT and exits. The bug was fixed by the developer.

cjpeg (libjpeg7). The program reads a zero value from the input file and then uses it as a divisor without checking it in a proper way; as a result a floating point exception occurs, and the program receives SIGFPE and exits. The bug was fixed by the developer.

sndfile-mix-to-mono (sndfile-tools-1.02). As in the 2 2
preceding program, division by zero is performed. The developer informed us that the bug has been already fixed in the current development version.

swfdump (swftools-0.9.0). Both bugs are null pointer dereferences.

avibench (avifile). Null pointer dereference.

xmllint (libxml2-2.7.6). No bugs detected.

mpeg2dec (libmpeg2-0.5.1). As in cjpeg and snd- 2
file-mix-to-mono, the detected bug is a possible division by zero.

mpeg3dump (libmpeg3-1.8). Both bugs are null pointer dereferences.

speexenc (speex-1.2rc1). No actual defects were found. Avalanche reports about 24 detected bugs because the timer expires (300 seconds) when application is run with each of these files. However, in none of the cases the loop is actually infinite; therefore, we can consider the reported defects as false positives.

7. RELATED WORKS

The general principles of Avalanche (generation of test cases in order to cover new fragments of program, the strategy for traversing the conditional jumps tree and increase of coverage metrics) are similar to those used in the SAGE tool described in [2]. However, in contrast to Avalanche, SAGE does not search the bugs purposefully; it is rather focused on traversal of as large

part of the conditional jumps tree as possible. In Avalanche, the purposeful reproduction of bugs is achieved by creating and checking constraints for dangerous operations. Other significant distinctions of Avalanche from SAGE are as follows.

• SAGE generates constraints while processing previously collected execution trace saved in the form of a sequence of x86 instructions. Avalanche generates constraints during the execution of dynamically instrumented program. The use of the Valgrind intermediate representation as an input language for translation into STP declarations significantly simplifies the constraint generation. Simultaneous constraint generation and program execution considerably speeds up the analysis.

• Avalanche has a much higher prediction accuracy (for SAGE, the prediction accuracy does not exceed 40%).

• Avalanche allows specifying a threshold value for the lookup depth.

The tool EXE described in [3] is also similar to Avalanche. A significant feature of EXE is the instrumentation of the source code (in contrast to the instrumentation of the executable binaries in the case of Valgrind) and requirement to manually mark up all the sources of tainted data in the program. One of the main advantages of EXE is the easier generation of constraints for checking various dangerous operations. For example, EXE may generate constraints for checking buffer overflows because both the array length and the index are always explicitly specified in the source code.

Like Avalanche, Catchconv [6] uses Valgrind–STP chain for collecting constraints and checking their satisfiability. However, Catchconv generates constraints only for checking a relatively narrow class of bugs—signed and unsigned type conversion errors. Moreover, the analysis is preformed only on a single trace determined by the input data; no input data for traversing other possible traces is generated.

Flayer [4] is very different from Avalanche. Flayer also uses Valgrind for tracing the flow of tainted data in the program. However, Flayer does not generate and check any constraints; instead, it instruments the code in such a way that the conditional jumps are executed (or not executed) without actual checking of the corresponding conditions. This approach, firstly, requires the human involvement for controlling the analyzer, secondly, makes possible the appearance of false positives, and, thirdly, doesn't provide input data reproducing even the true positive defect.

Constraint checking is also used in static analyzers. The article [7] describes a static analysis tool that models the control flow, the data flow and all dynamic memory allocations using boolean formulas.

8. FUTURE WORK

The defects detected as the result of the analysis of several open source projects has proved effectiveness of Avalanche. However, the exponential complexity of the problems that need to be solved (satisfiability checking and traversal of the tree of conditional jumps) is a significant limitation. This limitation explains why only the bugs that are relatively close to the entry point of the program are detected effectively. For that reason, the development of techniques that make it possible to detect bugs in other parts of programs seems to be the most promising direction for further research.

In particular, we plan to explore the problem of parallelisation of different Avalanche components. Many stages of the analysis are completely independent of each other. In particular, all the constraints produced during the execution of a certain branch of the tree of conditional jumps can be checked independently; the non overlapping branches of the tree can also be executed simultaneously. This makes us hope that parallel work of different components of Avalanche can be organized effectively.

Furthermore, we are going to examine the possibility of performing partial analysis of programs preferably without considerable decrease in the accuracy of bug detection. It would be interesting to consider various variants of traversing separate fragments of the tree of conditional jumps. For example, functions in the program can be analyzed individually. Another possibility is to use heuristics to determine the most interesting (in the sense of defect detection) parts of the program. We also consider the suggestion of combining static and dynamic analysis: a static analyzer can somehow mark certain parts of the program which are to be then analyzed by Avalanche.

These methods promise to speedup Avalanche. Another interesting direction of development is improving the accuracy of bug detection. Currently, Avalanche performs a purposeful search (by checking the corresponding constraints using STP) of a rather narrow class of defects—division by zero and null pointer dereference. Obviously, the list of possible bugs contains bugs of many other types:

- buffer overflow;
- uninitialized variables;
- signed/unsigned type conversion errors;
- dynamic memory errors—memory leaks, double free, etc.;
- security vulnerabilities.

First, it is necessary to examine the problem of purposeful generation of conditions for checking the occurrence of these errors (like in the case of division by zero and null pointer dereferencing). There is a reason to assume that such conditions can be created for detecting signed/unsugned type conversion errors [6] and for certain cases of buffer overflow. Second, a tool for detecting such bugs need to be developed. (Recall

that presently Avalanche detects only those errors that lead to abnormal termination of the program. However, if the program contains a bug that does not lead to abnormal termination, such a bug is not detected.) In order to do that Avalanche should probably be integrated with other Valgrind tools.

Besides, the present functionality of Avalanche can be optimized. The list of supported sources of tainted data can be extended. Recall that currently only a single file can be a source of tainted data. In fact, there are many other sources; indeed, a program can work with several files, get information from command line arguments, or environment variables. All this data should be considered as tainted. Sockets can be an especially interesting source of tainted data. Support of sockets as a source of tainted data will enable Avalanche to analyze network applications, which will certainly make Avalanche even more useful and probably make it possible to detect new types of defects.

9. CONCLUSIONS

We described dynamic defect detection tool Avalanche. We explained the general principles of creating input data for the purposeful reproduction of errors and for the analysis of initially unreachable parts of program, described the strategy for traversing the conditional jumps tree, the heuristics used to select the branch for the further analysis, the model for translating Valgrind intermediate representation into STP declarations and its implementation as a Valgrind plugin.

We examined the results obtained by applying Avalanche for the analysis of open source projects and listed the detected bugs. We also indicated the main limitations that prevent the enhancement of the efficiency of bug detection and completeness of the analysis (the exponential complexity of algorithms for the traversal of the conditional jumps tree and satisfiability checking) and discussed directions of further research.

SPELL: 1. occurence, 2. sndfile

REFERENCES

1. Nethercote, N. and Seward, J., Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, 2007.
2. Godefroid, P., Levin, M., and Molnar, D., Automatic Whitebox Fuzz Testing, *Network Distributed Security Symposium (NDSS)*, 2008.
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., and Engler, D.R., EXE: Automatically Generating Inputs of Death, *Proc. of the 13th ACM Conference on Computer and Communications Security*, New York, 2006, New York: ACM Press, 2006, pp. 322–335.
4. Drewry, W. and Ormandy, T., Flayer: Exposing Application Internals, *First Workshop on Offensive Technologies (WOOT)*, 2007.
5. Ganesh, V. and Dill, D., A Decision Procedure for Bit-Vectors and Arrays, *Lect. Notes Comput. Sci.*, vol. 4590, 2007, pp. 519–531.
6. Molnar, D. and Wagner, D., Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors, *Technical report of the University of California, Berkeley*, 2007, No. 2007-23.
7. Xie, Y. and Aiken, A., Scalable Error Detection Using Boolean Satisfiability, *Proc. of the 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, Long Beach, California, 2005, pp. 351–363.