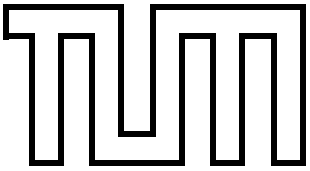


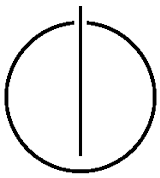
FAKULTÄT FÜR INFORMATIK

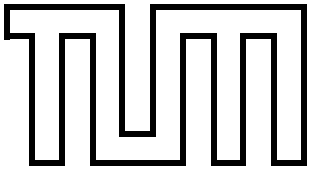
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Monkey - Generating Useful Bug Reports Automatically

Markus Teich





FAKULTÄT FÜR INFORMATIK

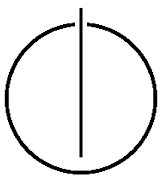
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Monkey - Generating Useful Bug Reports Automatically

Monkey - automatische Erstellung nützlicher Bugreports

Author:	Markus Teich
Supervisor:	Christian Grothoff PhD UCLA
Advisor:	Christian Grothoff PhD UCLA
Date:	July 15, 2013



I assure the single handed composition of this Bachelor Thesis only supported by declared resources.

München, July 15, 2013 Markus Teich

Acknowledgments

First of all I thank Christian Grothoff for his support and guidance throughout the creation of this Bachelor Thesis. He wrote the Cparser plugin and always had some good advice to give or code snippets to try when I was stuck.

I would also like to thank Safey A. Halim for his work and Master Thesis on which my work is based. The Pathologist is based on his version and his ideas for Seaspider and Entomologist have also been of great use.

For bits of code review I thank Kilian Röhner, Markus Otto, Stefan Winkler, Sven Hertle, Yang Hwan Lim and of course Christian Grothoff. I thank Veronika Ostler for helping me figuring out the weighting function in Entomologist.

I thank Christina for drawing the nice little Monkey Icon, which is used on the Monkey homepage.[1]

My thanks go also to Tom Tromey from the GDB developer mailing list for helping me figuring out the meanings of some tags, that can occur in GDB's output.

Additionally I thank my father Werner Teich, Felix Kampfer and Benedikt Peter for proofreading on short notice.

Last but not least I thank my family and friends for their great support.

[1] https://gnunet.org/monkey/

Abstract

Automatic crash handlers support software developers in finding bugs and fixing the problems in their code. Most of them behave similarly in providing the developer with a (symbolic) stack trace and a memory dump of the crashed application. This introduces some problems that we try to fix with our proposed automatic bug reporting system called "Monkey".

In this paper we describe the problems that occur when debugging widely distributed systems and how Monkey handles them. First, we describe our Motivation for developing the Monkey system. Afterwards we present the most common existing automatic crash handlers and how they work. Thirdly you will get an overview of the Monkey system and its components. In the fourth chapter we will analyze one report generated by Monkey, evaluate an online experiment we conducted and present some of our finding during the development of the clustering algorithm used to categorize crash reports. Last, we discuss some of Monkeys features and compare them to the existing approaches. Also some ideas for the future development of the Monkey system are presented before we conclude that Monkey's approach is promising, but some work is still left to establish Monkey in the open source community.

Contents

1 Motivation

When debugging widely distributed software systems, developers stumble upon four major problem domains.

For security reasons it often is not possible for developers to run a debugger on a system owned by end-users to reproduce a bug. Many users do not know how to setup a SSH server nor do they want to grant strangers any access rights on their machines. Therefore some systems have been developed, that automatically capture crashes on the users machine and create a crash report that can be sent to the developers in return. Normally the report is only sent with the users consent.

Secondly the system has to be very careful with private data. Depending on the verbosity of the report, there is a high chance, it contains private keys or data, that is not intended to be seen by developers that fix the bugs. For example I cought a Thunderbird crash capturing a core dump containing private emails. It I would have accepted to send the report to the Mozilla developers, I would have lost control over these emails the moment they leave my computer. I only found this, because I inspected the minidump file of the report in a hex editor. A novice user would not have noticed. Therefore it is important to let the user see **all** information that will be sent in a readable way.

Also many developers find, they get too many crash reports to process effectively. Therefore some try to keep the programm running as long as possible even if in an unstable state, remove assertions and if they use an automatic bug reporting system, they only run it in the most critical cases. This however is the wrong approach. Developers should be encouraged to include assertions to check critical conditions whenever possible, because a bug is much easier to fix if you get an assertion failure report of the condition that caused the bug rather than an uninformative report of where the application finally could not handle the incorrect state anymore and crashes in possibly random locations.

This would lead to more reports, but developers would probably still not be able to handle this additional amount of reports. Therefore a technique must be found to filter out the most important reports to be fixed as early as possible. If there is a common

bug that happens every time the application starts, then this one should be fixed first of course. A crash that occurs on machines of half the userbase should be fixed before looking into a very specific crash that only occurs on a few machines. Typical automatic bug reporting systems mostly report just a superficial stacktrace or a whole core dump, packed with some additional system information. Both data formats are not convenient for comparison, so it is hard to detect similar crash reports automatically. Because of this lack of useful sorting techniques, often two developers work on two different crash reports but both correspond to the same bug. This wastes time, that could have been spent on debugging other crashes or writing new code.

2 Related Work

In this section we discuss some other bug reporting systems similar to Monkey, which all fulfill the following requirements:

- **Distributed**: The system has to generate reports automatically without developer interaction.
- **Information**: The system has to provide a stack trace, dump, or something similar to help the developer find the bug.
- **Submission**: The system has to provide a simple way to submit reports to the developer.
- **Analysis**: The system has to contain some sort of report organization, filtering or comparison.
- **Viewer**: The system must supply reports in a way easily accessible to humans.

2.1 Windows Error Reporting

Since Windows XP, Windows Error Reporting (WER) has been introduced as crash handler for the Windows operating system (OS) and is shipped with it by default. It recognizes crashes of the Windows kernel, Windows components and other applications. A report containing system information and a memory-dump is generated post-mortem and the user can choose to send it to Microsoft. On their servers it is analyzed, categorized and if there is already a solution to the problem, it is suggested to the user via WER response. In Windows Vista a new application programming interface (API) has been introduced to allow developers to modify and add additional information to the reports concerning their applications and even generate own reports. However all reports are still sent to the Microsoft servers, allthough it is is possible for developers to get access to them if they have a VeriSign Class 3 ID.

Due to the proprietary nature of Windows we can not know how WER exactly works and what tools are used by Microsoft to sort and group the reports.

2.2 Apport

Ubuntu application crashes are detected and handled by Apport. When a user-space program caught a bad signal, like SIGSEGV, or an unhandled exception occurs (from e.g. a Python application), Apport collects the current execution status, OS environment information, packaging details, stack frame, core dump and further details about the crash itself. A dialog appears where the user can review the collected data and send the report to the corresponding bug tracker. Identical bug reports can be averted if the bug can be identified uniquely only with regular expressions on the report. In this case the developer can send this manualy generated bug pattern to the Ubuntu bug control team to append it to their publicly available bug pattern database. Example patterns for two specific bugs are shown in Figure 2.1. To easen the analysis of the report for the developer, the stack frame can automatically be extended to a backtrace by downloading the relevant debug packages.

Developers can also add additional information, exclude certain code paths from error detection and even run custom code like surveys or cleanup scripts through Apport hooks.

2.3 Google Breakpad

Google Breakpad is a multi-platform library which handles crashes on application-level and is used in Google Chrome, Mozilla Firefox and other applications. It helps the developer to get a dump keeping the interaction with the crashed thread as low as possible [3]. The developer has to include the library and implement some callback functions where he can collect additional information. The report is sent via HTTP(S) POST to a server chosen by the application developer.

During the writing of this report I encountered a Thunderbird crash, which resulted in a minidump file containing private email contents. These minidump files can be sent to the server, where they are processed and can be viewed in a webinterface afterwards. During processing a symbolic stack trace is generated out of the minidump file and developer supplied symbol files, so the application does not need to ship additional symbol information [4]. No more private data should be contained in the symbolic stacktrace. However, the user is still sending a whole dump over the network and henceforth has no control over it's content furthermore after sending in the report.

Figure 2.1: Example patterns for two specific bugs in Apport

```
<pattern url="https://launchpad.net/bugs/810182">
  <re key="Package">^nux-tools </re>
  <re key="Title">unity_support_test crashed with SIGSEGV</re>
  <re key="Stacktrace">get_opengl_version</re>
</pattern>

<pattern url="https://launchpad.net/bugs/833348">
  <re key="Package">^compizconfig-settings-manager </re>
  <re key="Title">ccsm crashed with KeyError in compizconfig.Plugin.ApplyStringExtensions
      ↪ \(src/compizconfig.c:6780\)\(\):</re>
  <re key="Traceback">File "/usr/lib/python2.7/dist-packages/ccm/Utils.py", line 328, in Update
  if self.Context.ProcessEvents\(\):</re>
  <re key="Traceback">File "compizconfig.pyx", line 1163, in compizconfig.Context.ProcessEvents
      ↪ \(src/compizconfig.c:9986\)</re>
  <re key="Traceback">File "compizconfig.pyx", line 1249, in
      ↪ compizconfig.Context.ChangedSettings.__get__ \(src/compizconfig.c:11104\)</re>
  <re key="Traceback">File "compizconfig.pyx", line 447, in compizconfig.SettingListToList
      ↪ \(src/compizconfig.c:2212\)</re>
  <re key="Traceback">File "compizconfig.pyx", line 927, in compizconfig.Plugin.Screen.__get__
      ↪ \(src/compizconfig.c:7371\)</re>
  <re key="Traceback">File "compizconfig.pyx", line 783, in compizconfig.Plugin.Update
      ↪ \(src/compizconfig.c:5765\)</re>
  <re key="Traceback">File "compizconfig.pyx", line 870, in
      ↪ compizconfig.Plugin.ApplyStringExtensions \(src/compizconfig.c:6780\)</re>
  <re key="Traceback">KeyError:</re>
</pattern>
```

2.4 Full Backtrace

A full backtrace created by GDB with the command `bt full` can also be seen as a bug report, however it may contain private data, since all local variables are contained. Also it would not be very helpfull if the application crashes on a line like `*(a[b]->c) = d;`.The full backtrace would just contain values for `a`, `b`, `c` and `d` but not for, e.g., `a[b]->c`.

3 System Architecture

3.1 Requirements

To use the Monkey Debugging System, you have to fulfill the following requirements:

- Your project has to be buildable with Cparser.
- You need GDB in version 7 or later.

3.2 Overview

The Monkey Debugging System consists of three parts, which work as a tool chain in a strictly linear fashion. An overview of the workflow can be seen in Figure 3.1. For a detailed description of the different components see the following sections.

3.3 Static Code Analysis

To allow the dynamic evaluation of possibly interesting expressions after the client program crashes, we first have to get a list of these expressions from the source code of the client. Since every expression in the abstract syntax tree (AST) is of possible interest, we use the C compiler "Cparser" with a custom addon to get every expression in the AST and leave the decision which ones to evaluate to Pathologist. The Cparser addon is generating a text file with the file name suffix `.sea` per compilation unit, which contains the corresponding expressions. An example program can be seen in Figure 3.2 and the output for this program generated by Cparser in Figure 3.3.

The `.sea` file does not just contain a list of expressions, but also some additional meta data. We get the source file name and the line number of each expression. If the last field in a `Begin-Expression` line is set to `1`, then the following expression contains a function call. Also, we note the beginning and end of each scope. This meta data is used in the Python script "Seaspider" to generate an expression database (EDB). Based on the content of this database Pathologist can decide, which expressions to evaluate.

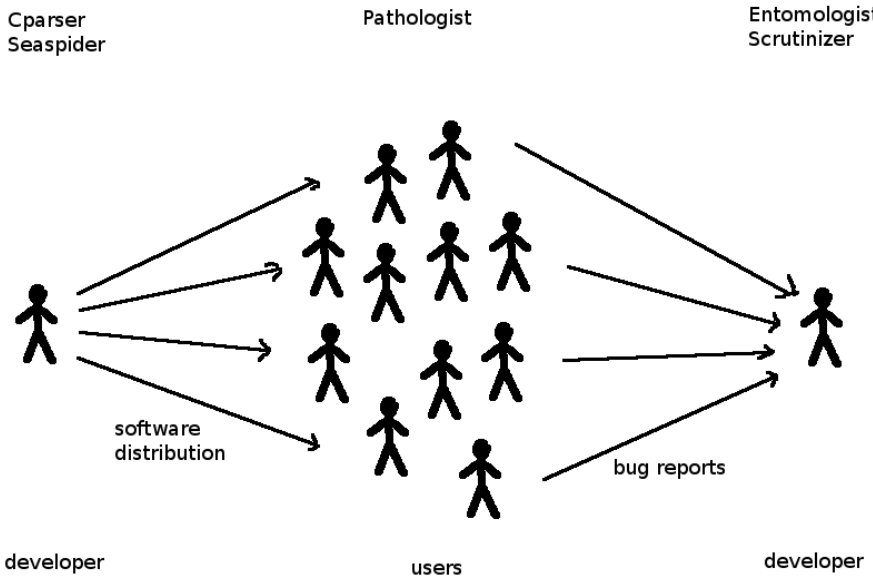


Figure 3.1: Monkey's system architecture

This database is in sqlite format, because we can deliver a single file easily. Giving end users access to e.g., an online database would incur lags during debugging or require the user to run his own database server. The sqlite database is generated from all the `.sea` files in a project with seaspider. Seaspider also allows the developer to specify some regular expressions and every expression that matches, will not be added to the database. In this way we can ensure that no expressions containing private, sensitive or cryptographic data get evaluated and added to the report. The database consists of a single table with the schema shown in Figure 3.4.

From the given meta data we calculate the line range in which each expression is valid, so we do not need to save functions or scopes in the database. Globals will get a range from the line where they are declared until line `32767`, so your source files should not reach that line count. Our previous example program would result in the sqlite Table 3.1.

This EDB contains all expressions from the syntax tree, except the ones containing a function call or being explicitely excluded by the developer. Each expression is also tagged with the surrounding context line numbers and file name to allow Pathologist to retrieve only the relevant expressions from the EDB.

Figure 3.2: Source code in example.c

```
#define MAXFAC 10

int calc(int v) {
      if(v < 2) return 1;
      return v * calc(v - 1);
}

int main(int argc, char* argv[]) {
      return calc(argc) < MAXFAC;
}
```

3.4 Debugging and Report Generation

After the EDB is generated by the developer, he can ship his application bundled with Pathologist and the EDB to the user base. Instead of running the application directly, the user runs Pathologist. Pathologist has to be set up with certain command line arguments in order to find the EDB and client binary to debug. Also, some features can be enabled or disabled.

Pathologist starts the client programm through libgdbmi[1], which handles all the communication with GDB. Pathologist then waits for the client to either exit gracefully or crash. When one of the supported signals is caught, Pathologist analyzes the stack through GDB, extracts the relevant expressions from the EDB, evaluates these expressions, and then adds them to the JSON-formatted report.

After we have the location of the statement that caused the crash, we can extract every relevant expression from the EDB. Every expression that is valid and accessible at the crash location is considered relevant. This means that we can run a simple select statement on the EDB which returns all expressions that begin to be valid before or at the crash location and stop being valid after the crash location. Note that we exclude globals (marked with an `end_lineno` of `32767`) because they will be put in a different section in the report. We collect this list of expressions for each stack frame and evaluate them with GDB afterwards.

When a crash is detected, Pathologist also runs a customizable system information script parallel to detect system time, OS version and other information. If you write your own script, it also has to be distributed with Pathologist.

[1] http://sourceforge.net/projects/libmigdb/

Figure 3.3: Output from Cparser in example.sea

```
1  Function: example.c:3 calc
2  Parameter: example.c:3 v
3  Scope: example.c:3 5
4  Begin-Expression: example.c:4 0v
5  End-Expression: example.c:4 0
6  Begin-Expression: example.c:4 0
7  2
8  End-Expression: example.c:4 0
9  Begin-Expression: example.c:4 0
10 v < 2
11 End-Expression: example.c:4 0
12 Begin-Expression: example.c:4 0
13 1
14 End-Expression: example.c:4 0
15 Begin-Expression: example.c:5 0
16 v
17 End-Expression: example.c:5 0
18 Begin-Expression: example.c:5 0
19 calc
20 End-Expression: example.c:5 0
21 Begin-Expression: example.c:5 0
22 v
23 End-Expression: example.c:5 0
24 Begin-Expression: example.c:5 0
25 1
26 End-Expression: example.c:5 0
27 Begin-Expression: example.c:5 0
28 v - 1
29 End-Expression: example.c:5 0
30 Begin-Expression: example.c:5 1
31 calc(v - 1)
32 End-Expression: example.c:5 1
33 Begin-Expression: example.c:5 1
34 v * calc(v - 1)
35 End-Expression: example.c:5 1
36 Function: example.c:8 main
37 Parameter: example.c:8 argc
38 Parameter: example.c:8 argv
39 Scope: example.c:8 9
40 Begin-Expression: example.c:9 0
41 calc
42 End-Expression: example.c:9 0
43 Begin-Expression: example.c:9 0
44 argc
45 End-Expression: example.c:9 0
46 Begin-Expression: example.c:9 1
47 calc(argc)
48 End-Expression: example.c:9 1
49 Begin-Expression: example.c:9 0
50 10
51 End-Expression: example.c:9 0
52 Begin-Expression: example.c:9 1
53 calc(argc) < 10
54 End-Expression: example.c:9 1
```

Figure 3.4: SQL schema of the EDB

```
CREATE TABLE Expression (
      expr_ID INT PRIMARY KEY AUTOINCREMENT,
      file_name TEXT NOT NULL,
      expr_syntax TEXT,
      start_lineno INT,
      end_lineno INT,
      is_call INT,
      UNIQUE (file_name, expr_syntax, start_lineno,
         ↪ end_lineno))
```

The report genererated by Pathologist is formatted in JSON. This has some advantages over the previously used XML format. The overhead of the JSON format is considerably smaller than the overhead of the XML format. This saves space on the developer's report storage server and the reports can also be transferred faster from the user to the developer. Additionally the specification is considerably smaller and since the developer should write his own script to collect system information and this script is expected to output valid JSON markup, the JSON format is to be preferred. The JSON generation and parsing libraries are also smaller due to their simplicity, and therefore less error prone. A small drawback of the JSON format is its missing support for attributes on nodes. This, however, is not a needed feature in the current report format, which you can see in Figure 3.5. An example report is also attached in the appendix.

3.5 Report Comparison and Clustering

After we have collected some crash reports from the users, the bugs will still persist, until we actually fix them. To help developers handle an overwhelming amount of crash reports, we provide Entomologist, a Python script which carries out K-means clustering on a given set of reports.

To apply K-means clustering on our report set, we first need to define a distance function which takes two reports and outputs a scalar representing the distance or amount of differences between the two reports. Since the work of Safey[2], I rewrote Entomologist in Python. Existing general tree structure comparison algorithms do not seem to be promising for bug report comparison. Therefore I propose a new distance heuristic as described[2] in Algorithm 1. For the function dexp, which compares two lists of

[2]We write $A.trace$ directly instead of $A.history[0].trace$

Table 3.1: Expression database

ID	file name	syntax	start	end	is call
1	example.c	expected	3	32767	0
2	example.c	v	5	8	0
3	example.c	v	6	8	0
4	example.c	2	6	8	0
5	example.c	v < 2	6	8	0
6	example.c	1	6	8	0
7	example.c	v	7	8	0
8	example.c	calc	7	8	0
10	example.c	1	7	8	0
11	example.c	v - 1	7	8	0
12	example.c	calc(v - 1)	7	8	1
13	example.c	v * calc(v - 1)	7	8	1
14	example.c	argc	10	13	0
15	example.c	argv	10	13	0
16	example.c	fac	12	13	0
17	example.c	10	12	13	0
18	example.c	fac > 10	12	13	0
19	example.c	1	12	13	0
20	example.c	-1	12	13	0
22	example.c	expected	12	13	0
23	example.c	fac != expected	12	13	0
24	example.c	fac > 10 ? -1 : fac != expected	12	13	0

expressions, see Algorithm 2.

To divide the given set of crash reports into different clusters, we use a modified version of the "Far Efficient K-Means Clustering Algorithm"[5]. The originally proposed algorithm depends on calculating the mean of a set of elements to get the centroids. However, it is difficult to generate a new crash report that lies exactly in the middle between two other reports. My solution is instead to use a representative system to calculate the centroids. Whenever we need to find a new centroid for a given set of reports, we just pick the report that has the minimal sum of distances to each report in the set. This should not affect the result of the algorithm, since we do not expect bugs, which result in circular clusters, where each report would have aproximately the same distance to one central report, that, however, does not exist. Also, after the execution of the algorithm, we can just output the list of reports that were found to be the centroids of the clusters. These reports are suggested to the developer to look into to fix the most common bugs.

Figure 3.5: The format of the reports generated by Pathologist

⟨*report*⟩ ::= '`{`' ⟨*category*⟩ '`,`' ⟨*sysinfo*⟩ '`,`' ⟨*history*⟩ '`}`'

⟨*category*⟩ ::= '`"category":"`' ⟨*cat*⟩ '`"`'

⟨*cat*⟩ ::= '`bad memory access`'
| '`null pointer access`'
| '`bus error`'
| '`assertion failure`'
| '`arithmetic exception`'

⟨*sysinfo*⟩ ::= '`"sysinfo":`' ⟨*JSON_object*⟩

⟨*history*⟩ ::= '`"history":[`' ⟨*step*⟩ '`]`'

⟨*step*⟩ ::= ⟨*step*⟩ '`,`' ⟨*step*⟩
| '`{"trace":[`' ⟨*frame*⟩ '`],"globals":{`' ⟨*expr*⟩ '`}}`'

⟨*frame*⟩ ::= ⟨*frame*⟩ '`,`' ⟨*frame*⟩
| '`{"function":`' ⟨*JSON_string*⟩ '`,"line":`' ⟨*JSON_number*⟩ '`,"file":`' ⟨*JSON_string*⟩ '`,"expressions":{`' ⟨*expr*⟩ '`}}`'

⟨*expr*⟩ ::= ⟨*expr*⟩ '`,`' ⟨*expr*⟩
| ⟨*JSON_string*⟩ '`:`' ⟨*JSON_value*⟩

The second modification deals with the fact that we expect some unusual crash reports, that are not like any other and therefore should end up in an own or a small cluster. The original algorithm uses a special initialization step, where we first find the two most distant reports and then add a threshold of the $0.5 * n/k$ nearest reports to both centroids with n being the number of reports to cluster and k being the count of clusters we want to generate. Then the centroids of this two clusters are recalculated. Now, for each remaining cluster to initialize, a first guess of the centroid is searched for in the remaining data set, the same count of nearest threshold reports is added and the centroid is recalculated. In the case of a single report that does not have any related reports that should end up in its cluster, this report has above average distance to all reports and therefore probably ends up in one of the first centroids. Then we would add $0.5 * n/k$ reports to it during the initialization and get a pretty bad representative after recalculating the centroid. Since the quality of k-means clustering strongly depends on the initial set of clusters choosen, we change this initialization in the following way:

We leave out the adding of the threshold reports and just get a set of centroids. On our testset of 70 reports from 5 bugs, this produced better results than the extended initialization method in average.

3.6 Report viewing

Monkey also ships with a GUI to view reports called Scrutinizer. It is implemented in HTML and JS and runs in any modern web browser without a webserver required. Scrutinizer imitates the look of an IDE, as you can see in Figure 3.6. On the left you can browse the syntax highlighted source code. On the top right a linked stack trace can be seen, which automatically opens the correct source file at the respective line. On the bottom right we have the bug category and a list of all expressions that are valid in the currently selected stack frame.

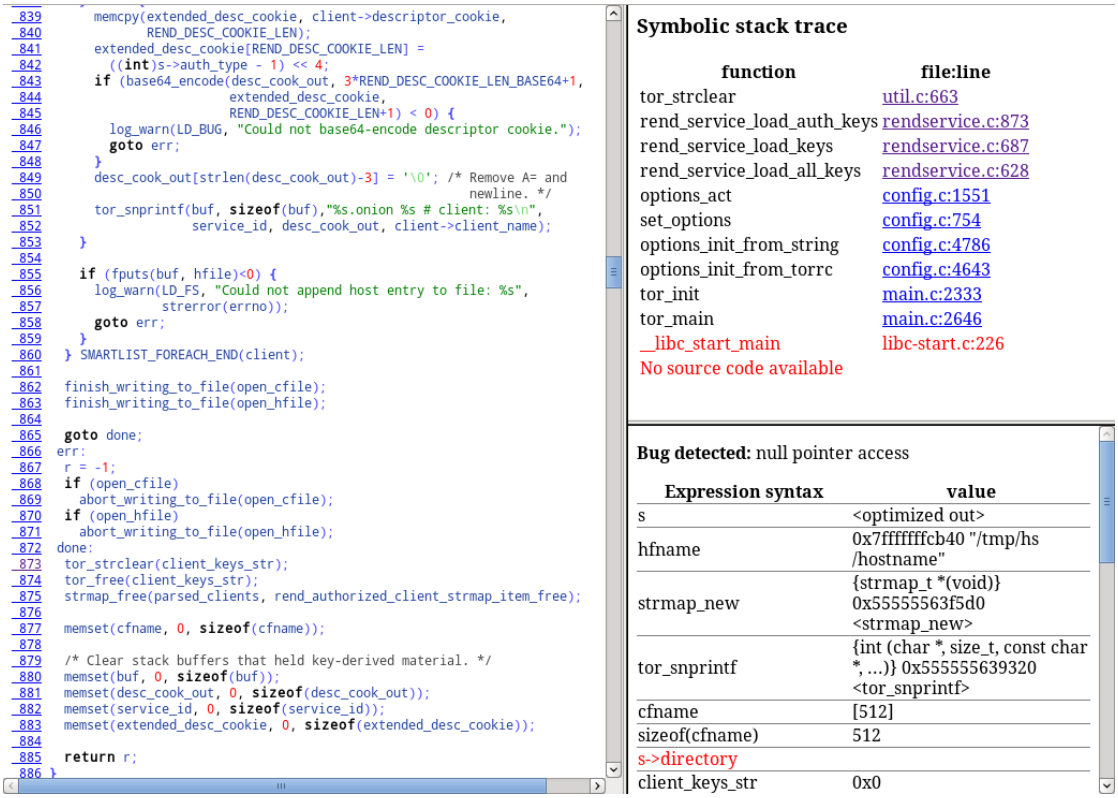


Figure 3.6: A screenshot of Scrutinizer

3.7 Limitations

- We have only tested the system on GNU/Linux and for one bug on FreeBSD, but it should easily be portable to any UNIX like system.
- Cparser mostly does not work on 64bit systems.
- Pathologist can only (nicely) handle the signals noted in Table 3.2.
- Pathologist does not forward incoming signals to the client, except SIGINT.
- Your C source file names have to be globally unique in your project, because GDB sometimes gets the whole relative path in the stack frame and sometimes just the file name depending on the build system.
- Due to GDB changing the respective values on the stack while evaluating, increments and decrements are problematic if contained in the line of code which causes the crash. Seaspider will just remove every single increment and decrement operator before writing expressions to the database. Therefore the report will only contain expressions with the variable to increment/decrement and not with the incremented/decremented value.

input : Reports A and B
output: Distance $dist$ between A and B

$f_{category} \leftarrow 20.0$;
$f_{function} \leftarrow 5.0$;
$f_{file} \leftarrow 5.0$;
$f_{line} \leftarrow 5.0$;
$f_{topframe} \leftarrow 4.0$;
$f_{frames} \leftarrow 20.0$;
$f_{frameexpr} \leftarrow 0.6$;
$f_{frameline} \leftarrow 0.4$;
$dist \leftarrow 0.0$;
$tw \leftarrow 0.0$;
$sum \leftarrow 0.0$;
$usedB \leftarrow \emptyset$;
if $A.category \neq B.category$ **then**
  $dist \leftarrow dist + f_{category}$;
end
if $A.trace[0].function \neq B.trace[0].function$ **then**
  $dist \leftarrow dist + f_{topframe} * f_{function}$;
end
if $A.trace[0].file \neq B.trace[0].file$ **then**
  $dist \leftarrow dist + f_{topframe} * f_{file}$;
end
if $A.trace[0].line \neq B.trace[0].line$ **then**
  $dist \leftarrow dist + f_{topframe} * f_{line}$;
end
for $f_i \in A.trace$ **do**
  for $g_j \in B.trace$ **do**
    if $j \notin usedB \wedge f_i.function = g_j.function \wedge f_i.file = g_j.file$ **then**
      $weight \leftarrow |A.trace| + |B.trace| - i - j$;
      $sum \leftarrow sum + weight$;
      if $f_i.line = g_j.line$ **then**
        $tw \leftarrow tw + weight * f_{frameline}$;
      end
      $tw \leftarrow tw + weight * f_{frameexpr} * dexp(f_i.expressions, g_j.expressions)$;
      $usedB \leftarrow usedB \cup \{j\}$;
      break;
    end
  end
end
if $sum \neq 0$ **then**
  $dist \leftarrow dist + tw/sum$;
end
return $dist$;

Algorithm 1: The Distance Heuristic of Entomologist

input : Expression lists a and b
output: Distance $dist$ between the two lists in range $[0, 1]$

$sum \leftarrow 0.0$;
for $el \in a$ **do**
  if $el \notin b$ **then**
    continue;
  end
  if $isNumber(a[el]) \wedge isNumber(b[el])$ **then**
    if $(a[el] \in \{0, 1, -1\} \vee b[el] \in \{1, 0, -1\}) \wedge a[el] \neq b[el]$ **then**
      $sum \leftarrow sum + 1$;
    end
  else if
  $isString(a[el]) \wedge isString(b[el]) \wedge a[el].startswith('0x') \wedge b[el].startswith('0x')$
  then
    $a \leftarrow int(a[el], 16)$;
    $b \leftarrow int(b[el], 16)$;
    if $a = 0$ **then**
      $x \leftarrow 0$;
    else if $a < 1024$ **then**
      $x \leftarrow 1$;
    else
      $x \leftarrow 2$;
    end
    if $b = 0$ **then**
      $y \leftarrow 0$;
    else if $b < 1024$ **then**
      $y \leftarrow 1$;
    else
      $y \leftarrow 2$;
    end
    if $x + y = 2$ **then**
      $sum \leftarrow sum + 1$;
    else if $x \neq y$ **then**
      $sum \leftarrow sum + 0.5$;
    end
  else
    if $a[el] \neq b[el]$ **then**
      $sum \leftarrow sum + 1$;
    end
  end
end
return $sum * 2/(|a| + |b|)$;

Algorithm 2: The Distance Heuristic for two expression lists

Table 3.2: Signals handled by Pathologist

No.	Signal name	Signal meaning
6	SIGABRT	Assertion or call to abort()
7,10	SIGBUS	Bus error
8	SIGFPE	Floating point exception
11	SIGSEGV	Segmentation fault or bad memory acces

4 Experimental Results

4.1 Report Analysis

Here we evaluate one report generated by version 0.2.1 of Pathologist attached in Section 6.2 in the Appendix. The crash occured in tor and you can find details about this bug in the official bugtracker.[1]

The contents of the `sysinfo` tree in line 3 are fully customizable by the developer. Monkey is shiped with an example script, that generates the attached system information tree. You probably want to replace `ls` with your own binary name at least, to see which library versions are used. Also, it probably would help to add an indicator for the version of the client program that is debugged. Be careful though not to add any output that could contain private data of the user.

The `history` array beginning in line 19 does only contain one element in this report, since we did not record any execution steps. Inside we have two elements, the `global` node, which contains a list of expressions from the global namespace, and the `trace` node, which contains the stack trace. Since there were no global expressions valid in this crashes context, this node got assigned the value `null`. If there were any global expressions, the global node would look exactly the same as the `expressions` node in the stack frames.

The `trace` node gives us the `function` name, `line` of code, `file` name and the list of `expressions` for each stack frame. Since function names must only be unique in one compilation unit, we have to include the file name to the report too. Otherwise, there could be conflicts with two equally named static functions, which are only accessible from their respective source file. Entomologist would not know about this without the file name information included. In the expression list we can see some values that need explanation.

In line 25 we have an expression called `net_params` which is obviously a pointer, because the associated value is a hexadecimal number. The exact memory address is not very informative from a developer's point of view. However, we can not just evaluate

[1] https://trac.torproject.org/projects/tor/ticket/8553

the value the expression points to, since it could contain private data, which we can not easily exclude from the report by expression names. Consider a generic function to insert an element in a linked list. The data to insert would probably be a pointer which is not dereferenced in this function. Therefore, we would only have the address in our report if the crash occurs in this function. If we would carelessly evaluate the memory pointed to, we could leak private data. The only thing we can improve about this is looking up the memory segment (stack, heap, libraries memory, read-, write- and execute properties) it points to in `/proc/PID/maps`. See Section 5.2 (Future Work) for a further note on this topic.

In line 30 we see an expression evaluated to `null`. This does not represent a NULL Pointer,[2] but an expression that cannot be evaluated, because it has been optimized out by the compiler or is a pointer to an inlined function. If you get lots of these but need to know their values, consider adding `-O0` to the `CFLAGS` compiler options to disable optimizations completely.

From line 31 to 34 there are several expressions that only differ in the amount of exclamation marks in front of them. This is caused by the static code analysis step, where each preprocessor macro gets expanded and the `tor_assert` macro uses up to triple negation of the expression, so every one is included in the report. This could be prevented by adding fitting ignorefile entries like `::!!(.*)` before running the Seaspider script.

In line 35 we see an evaluated function pointer in the report. This is not always useful, but can greatly help debugging a function which takes a callback function as argument. In this case you probably want to know, which callback function has been associated with this function call.

The expressions from line 36 to 40 are all not particularly helpful. For line 38 and 40 you could add entries to the ignore file, since they only contain the function name which we already know. For the other three lines Cparser would have to know that they are constants and therefore not of interest and can be excluded from the EDB. At the moment, at least Pathologist checks if the expression syntax is the same as the result of the evaluation to strip some useless expressions from the reports. However this should be done as early as possible, that is during the static code analysis by Cparser.

In the second stack frame we see two empty string values in lines 55 and 56. These are caused by the fact, that `options` has been optimized out as you can see in line 54. Therefore, obviously, also subexpressions, that depend on `options` can not be evaluated.

[2] Null pointers are actually represented as "0x0"

In the last two stack frames beginning at line 131 we can see that there are no expressions in the list. Since their source code was not located in the main tor source code repository, there has not been any static code analysis on them. The last one, `_start`, even is the entrypoint to the binary, which gets added by gcc and therefore could not possibly have any data from Cparser. So this is no problem with the report.

4.2 Survey

To evaluate, how well the Monkey reports perform in comparison to existing report formats, we started an online Experiment. Every participant got to see three different crashes and each one represented by a randomly selected crash report format.

The three crashes were always presented in the same order, sorted by our estimation of how hard it would be to find the bug. The first one was the easiest, where just two preprocessor defines (`DFLT_GUARD_LIFETIME` and `MIN_GUARD_LIFETIME`) were switched, which caused an assertion failure in a function one level deeper in the stack trace. To find this bug, the developer had to navigate one stack frame up and scroll up the source file a few lines above the function from this stack frame, where the malformed definition can be found. The second bug is an unchecked NULL pointer beeing passed to a function `tor_strclear`, which is responsible to zero out the given string before it gets freed in another function. A possible fix would have been to remove the call to `tor_strclear`, since the `tor_free` called afterwards checks for NULL pointers. However this would induce serious security problems for strings which contained cryptographic material. If they would be freed carelessly, this data would still be available in memory. The correct fix therefore is just to check for NULL pointers before calling `tor_strclear`. The third bug was the hardest to find, especially if the participant only got the report limited to a stack trace. The crash is a bad memory access caused by reading an array entry with an invalid index. The variable containing this index was assigned in a function call, that is not contained in the stack trace and even if the participant found this assignment, the error is still hard to see, because it involved a preprocessor macro, which mixed signed with unsigned int calculations which lead to a buffer underflow and therefore to the unreasonably high array index. The correct fix was to change the type of one of the variables used in this calculation from unsigned int to signed int. You can find all three crashed projects (tor-04, tor-03, st-01) in our `monkey-bug-db` git repository.[3] For each crash report the user was asked to find the bug and then click next. On the next page (without the possibility to go back) the user

[3] https://gnunet.org/monkey/git

was presented with three answers, where only one is correct and one option "I did not find the bug".

Apart from a **Monkey** report, usable through Scrutinizer, one report format consistet of the output of GDB's `backtrace` command, the **stacktrace** report, and the third report format additionally contained a live GDB session on a saved **core dump** of the crashed application. Although the two formats other than Monkey's are not exactly the same as what you get from e.g. Apport or Breakpad, they basically contain all information that is used by a typical developer to find the bug. The whole experiment also emulates the scenario where the developer is not able to reproduce the bug and therefore the only information he has is the report and the source of the project, which also has been included in every report format as a link that opens the browsable source code in a new browser tab. Every participant got to see exactly one of each report format in a randomly selected way.

Each participant was also first asked the following questions to estimate their programming and debugging experience:

- **How many years of experience in programming do you have?** Possible answers were "0 - 5 years", "5 - 10 years", "10 - 20 years" and "more than 20 years". Although the answer does not tell us anything about the actual skill levels of participants, at least we have a rough approximation of how familiar they are with the software development environment.
- **Have you contributed code to one of the following projects?** The participants could check any of "tor" and "st". Since crash reports to these two projects were shown later in the experiment, the answers to this question tell us how well the participants know these both software projects and if they maybe even encountered one of the bugs.
- **Which of the following bug reporting systems and debugging software have you used?** The participants could check any of "Apport", "Breakpad", "Monkey", "GDB", "Valgrind" and add other debugging tools to a text field if they have used any other. This question helps us to relate the response times for the three different report formats.

The result of the experiment can be downloaded from the Monkey website as a csv file.[4] Which reports were shown for which bug, depends on the `rand` number. You can find the assignment in Tabe 4.1. Here follows my interpretation of the results.

[4] https://gnunet.org/monkey/pub/survey-answers.csv

Table 4.1: rand number assignment

	tor-04	tor-03	st-01
backtrace	1, 4	3, 5	2, 6
core dump	2, 5	1, 6	3, 4
Monkey	3, 6	2, 4	1, 5

Table 4.2: correct and wrong answers per report format and bug (tor-04 + tor-03 + st-01)

	correct	wrong	did not find
backtrace	$6+3+0=\mathbf{9}$	$1+2+0=\mathbf{3}$	$1+3+5=\mathbf{9}$
core dump	$0+3+1=\mathbf{4}$	$1+1+2=\mathbf{4}$	$4+1+8=\mathbf{13}$
Monkey	$5+5+0=\mathbf{10}$	$1+2+1=\mathbf{4}$	$2+1+4=\mathbf{7}$

21 participants completed the experiment, which is not enough to conclude a generally valid result. Unfortunately the low number of participants in such experiments seems to be a general problem.[1] Looking at the correctness of the answers sorted by report system in Table 4.2, we see that the report including a core dump (but also the stack trace) got only half as much correct answers as the other two system. Although no statistical relevance is given, this may be caused by too much information beeing available. Since all except two participants have used gdb in the past and the only steps required to find the bug are to print out the two values in the assertion with GDB and figure out, where they come from, we can only assume, that too much information in the report distracts the developer from finding the bug. Especially on the first tor-04 bug the backtrace group got $6/8$ correct answers whilst having just a subset of the information from the core dump group which got $0/5$ correct answers. The core dump reports also have the highest amount of "I did not find it" answers with 13 out of 21.

Looking at the time, it took the participants to identify each bug with the three report systems in Table 4.3, we find, that for the first bug the participants that got a Monkey report only needed a third of the time, the participants with just a backtrace needed. Unfortunately none of the 5 participants getting a core dump for the tor-04 crash found the bug. There were no remarkable peaks in either dataset. For the second bug however

Table 4.3: average time it took to find the correct bug in seconds

	tor-04	tor-03	st-01
backtrace	304	202	No correct answer
core dump	No correct answer	274	364
Monkey	104	630	No correct answer

we get a slightly different picture. Whilst the backtrace group with an average of 202 seconds was around one fourth faster than the core dump group with 274 seconds on average, the Monkey report group took 630 seconds on average. This is caused by one participant taking the exceptional high amount of over half an hour to find the bug. Assuming this was caused by a disturbance during the experiment and removing this one data point, we get a new average of 317 seconds on average for the Monkey group. This is still slower than both other report formats, but not quite as much. For the third bug we can not conclude anything relevant, since only one participant found the bug using the core dump report after 364 seconds.

The three introductory questions were not taken into account[5] in this analysis, because on a dataset of 21 responses it is not justifiable to try to squeeze out the last bit of information, which probably does not hold for a wider range of people. In conclusion, we saw, that the Monkey reports can speed up the finding of bugs, but still many more responses would be required to make definitive statement.

4.3 Clustering

Whilst implementing the clustering algorithm, I noticed some things to be discussed in this section. All the clustering was done on a test set of 76 reports collected during the development cycle of Pathologist. Since the report format and content was changed a few times during this period, we have a variety of reports. You can also download[6] them to verify our results.

The reports are collected from 5 bugs in 2 different software packages. The first package is st[7], an x-terminal-emulator. Only one of the bugs is from st, it is a bad memory access failure in the indexing of an array caused by a mixed signed/unsigned integer calculation.[8]. The other 4 reports are all from Tor,[9] the anonymity software, and contain two null pointer accesses and two assertion failures. All 5 bugs are easily reproducible, the Tor bugs all occur during startup, for the st bug one has to click in the top pixel row of the window to cause the crash. Note also, that the two assertion failure bugs in tor occur in the exact same function, although on different versions of tor, on different lines and with different stack traces. This should make the problem of separating this two bugs in different clusters harder to solve but not impossible.

[5] they are included in the csv file nevertheless
[6] https://gnunet.org/monkey/pub/reports.tar.gz
[7] http://st.suckless.org/
[8] http://lists.suckless.org/dev/1211/13133.html
[9] https://www.torproject.org

During development of Entomologist we noticed that whilst calculating the perfect result (every report in a group with all the other reports from the respective bug) for $k = 5$ on one machine, on another machine we got another result. This is caused by the fact, that the Python function `lsdir` does not give the exact same order of reports. This leads to the case that the two initial reports selected are different on both machines and since the k-means clustering algorithm is highly dependent on the initial set of centroids, we get different results in the end.

5 Discussion and Conclusion

5.1 Features

Here we discuss the features of the Monkey system and compare them to features of existing systems. Also, we describe how the Monkey system should be developed in the future and conclude this Thesis.

5.1.1 Privacy

Many other automatic bug report generation systems may leak private data from the user. This can be caused due to the system's concept, that requires a core dump to be sent to the developer, or by other means like adding all CPU register values or a full backtrace to the report. Although in most cases the user can deny sending this detailed report in a popup dialog of some kind, the average user is not aware of this possible leakage. When I encountered the Thunderbird crash, I could not find any note on or trace of my personal emails contained in the report. Only after analyzing the minidump file with a hex editor, I found my private emails. Monkey could show the entire report as it would be sent to the developer. Neither binary encodings or dump files are used nor are the reports too long to skim over and detect such obviously critical content.

In the Monkey system it is not the user that has to decide which information may be privacy critical, but the developer. All relevant expressions can be removed preventively from the evaluation by Pathologist. The developer can even supply his own system information script which collects only absolutely relevant data. Due to this design, the Monkey system transfers responsibility of knowing, which data can be safely added to the reports, from the rather novice user to the developer, who should have the knowledge of his code to decide, which expressions to ignore. If the user is cautious, he can also inspect and modify the expression database to his likings since it is a plain sqlite file. If the process of excluding privacy relevant expressions from the EDB is done carefully, there should not be any private information leak within the Monkey system.

Since the Monkey system requires debug information to be included in the users binaries and most of the expressions from the source code can be read by the user from

the EDB, Monkey is probably not suited to debug closed source applications.

5.1.2 Completeness

Comparing the Monkey system with a simple full backtrace, we notice, that the full backtrace from GDB lacks the values of complex expressions like `a[index]->member`, which Seaspider extracts from the source code, so their values get added to the report later by Pathologist.

Currently, some expressions can not be evaluated by Pathologist. However, this is caused by compiler optimizations, which cause some variables to not exist in the binary and some functions to be inlined. This can be omitted by supplying the `-O0` option to GCC. Note, that every expression containing a variable that has been optimized out also can not be evaluated.

In comparison to a full core dump a Monkey report is not quite as verbose and limited to the expressions that are valid in the current stack trace. This seems to be the right direction and has a few advantages over a full core dump: The user has to transmit less data to file a bug report and the developer does not get distracted by mostly irrelevant content in the core dump. I think the Monkey report format does a good job at striking a balance between the superficial backtrace and the verbose core dump.

5.1.3 Size of the Distribution

The Monkey system needs to deploy two things appart from the client application to be debugged. First one being of course Pathologist, which should not exceed 2 MiB and secondly, the user needs an up to date version of the EDB for the respective client application. Depending on the client's code, the EDB can grow rather big, especially if it is using lots of preprocessor macros to expand code complexity even further. Generally nowadays it should not be problematic for network bandwidth nor for hard disk storage to support this extra load, but if the EDB is too big to the developers liking, he still can remove expressions from stable modules. This, however, has the disadvantage that he will not get a useful report if the case arises that one of this stripped modules catches a crash.

5.1.4 Crash Location

Whilst the WER reports the offset to the crash location in the binary, this is not a valid approach for UNIX software, because many distributions deliver different binary packages or even allow the user to compile them on their own. This would lead to untra-

cability of the exact crash location on the developers side. Monkey and other UNIX enabled crash detecting tools acquire the relevant source file and line number to describe the crash location. This does not include the exact assembler instruction causing the crash, but ensures portability and still provides a good enough clue of how the crash occured in most cases.

5.2 Future Work

For the future development of the Monkey debugging system the following ideas might be pursued.

5.2.1 Cparser / Seaspider

Since the static code analysis is still done in two steps, this could be simplified by adding the Seaspider script's functionallity directly into the Cparser addon. This should allow for faster generation of the expression database. You would have to rewrite the whole abstract syntax tree (AST) printing routine of Cparser, but you could also filter out unwanted expression types like increment operations directly while compiling and so you would not have to generate an intermediary text file.

At the moment, we can only track the crash itself, but in many cases the cause for the crash is located in a totaly different piece of code, that does not have to be in the stack trace when the crash occurs. If we would have a C compiler with support for control flow graph analysis on the AST layer, we could generate a more meaningful set of expressions to evaluate from the dominators of each instruction. This would also allow early removal of constant expressions from the EDB.

To ensure, that the stack is not modified during expression evaluation in Pathologist, we filter out all expressions containing function calls in Cparser. However, GDB would not modify the stack when calling pure functions, so Cparser could be extended to add pure function calls to the expression database.

For a wider spread usage of Monkey it could be beneficial to specify a system wide expression database, to which all expressions from client applications get added. However, this file could easily grow very big. Also, it probably has to contain a new field for application names to ommit duplicate source file names and allow for easy replacement of the fitting expressions when updating the client binary.

Support for other programming languages in the client application would further widen usability of the Monkey Debugging System. Since the grammar of C++ and other

languages is far more complex than the C grammar, it could be very hard to implement a static code analysis for these languages, which outputs an expression database like the one from Cparser/Seaspider.

5.2.2 Pathologist

In some occasions it would be useful to have the value of each CPU register in the report. Unfortunately, there is no easy way to filter out private data from this data source. Since we can not expect developers to specify which registers contain private data for each assembler instruction, this needs further research to find a useful solution.

Pathologist does not support debugging multithreaded applications currently. However, this feature would allow to apply the Monkey debugging system on many more client programs such as Qt based applications.[1] For this to work, Pathologist has to be modified to inspect the expression database in all threads and evaluate the resulting expressions. Also a feature to filter out irrelevant thread results would be nice.

A rather minor fix to implement would be to allow correct handling of client applications that require input over the stdin stream.

For the debugger it could be interesting to know the target memory type for each pointer in the report instead of the exact memory address. This information is available in `/proc/<PID>/maps`, but unfortunately the gdbmi library used in Pathologist does not provide the PID of the child process. To add this feature, the GDB command `maintenance info sections` could be added to libgdbmi. As an alternative one could add a breakpoint to main and call `print getpid()` with GDB there to get the GDB and then update the information from `/proc/<PID>/maps` every time the virtual memory is altered. This would require further breakpoints at those function calls like dlopen and mmap.

To allow more users and clients, it would first be useful to test Pathologist under Windows and Mac environments.

5.2.3 Entomologist

Although the initial constant factors for the distance metric turned out to be usable for our set of 76 reports, they probably should be customizable. It would be great to automatically adapt them to some feedback from the debugger. This would improve the results for the categorization of further reports into an existing cluster system.

[1] This implies using the C language bindings for Qt until C++ is supported by Monkey.

The clustering algorithm needs a good first guess for the centroids. Maybe we should run the algorithm with some random centroids a few times and pick the best result by the max in-cluster and min inter-cluster distances.

5.2.4 Scrutinizer

Scrutinizer needs some way to select which report to show. This should probably be integrated with Entomologist to allow fine tuning. The developer should be able to specify two reports are (not) the same bug.

5.3 Conclusion

In this Thesis I have shown that most currently used automatic bug reporting systems lack useable features to preserve the privacy of their users. Since there is no established way of categorizing crash reports in bug categories, some of these systems or single developers try to achieve less crash reports, which is definitely the wrong way. Developers should try to get as much reports as possible so they can decide which bug is the most urgent to fix.

The Monkey system adresses some of these problems through it's architecture and the results are promising. I cleaned up the Monkey code and made it work on real world applications like Tor. Clustering seems to be the right approach for categorizing crash reports. Monkey's reports viewed in Scrutinizer seem to help developers to find at least certain bugs faster than with existing report formats.

Still a software project is required to integrate Monkey in the development and debugging process to prove Monkey's stability and reliability in real world conditions. This is the next step to introduce Monkey to the open source debugging ecosystem.

6 Appendix

6.1 Tutorial

6.1.1 Prerequisites

To use the Monkey debugging suite you have to fulfill the following requirements:

- You need a 32bit machine to generate the expression database with Cparser/Seaspider.
- Your application has to be written in C and compilable with Cparser.

6.1.2 Installation

The following instructions assume you are using a new installation of a recent Debian/Ubuntu derivative. If you want to install Monkey on another distribution, make sure you run the commands prefixed with `sudo` as root and install the appropriate package dependencies yourself.

Cloning the repository

To install Monkey from the latest developer version, make sure, you have git installed and then clone the repository including the two required submodules: The extended Cparser available in our git and it's dependency libFirm.

```
$ mkdir ~/code && cd ~/code
$ git clone --recursive git://gnunet.org/monkey
$ cd monkey
```

Stable release

This step is optional. If you want to use the latest stable release, you can run:

```
$ git checkout $(git describe --abbrev=0)
$ git submodule update --recursive
```

Package dependencies

Monkey requires some packages installed before you can start the build process. To install them, run:

```
$ sudo aptitude install build-essential automake autopoint
   ↪ libtool libsqlite3-dev code2html gdb valgrind
```

Cparser/Seaspider installation

Now you can begin with the installation of Cparser and Seaspider.

```
$ cd seaspider/cparser
$ make
$ sudo make install_seaspider
$ cd ..
$ sudo make install
```

Pathologist installation

Next change to the Pathologist directory and install it, too:

```
$ cd ../pathologist
$ ./bootstrap
$ cd build
$ ../configure
$ make
$ sudo make install
$ sudo ldconfig
```

The last command might only be necessary on recent Debian systems due to a bug in the automake packages.

6.1.3 Example Project

Now you will create an example project to test Monkey and generate your first report.

```
$ cd ~/code
$ mkdir hsar
$ cd hsar
```

Now create the new source file `hsar.c` with your favorite editor and paste the following code:

```
#include <gcrypt.h>
#include <stdio.h>
#include <time.h>

#define GCRY_CIPHER GCRY_CIPHER_AES128
#define BLOCK_SIZE 16

int main(int argc, char* argv[])
{
  char* buf = malloc(BLOCK_SIZE);
  char* iv = "a test ini value";
  char* key = "one test AES key";
  clock_t start;
  FILE* fp = fopen("secret", "r");
  gcry_cipher_hd_t hd;
  size_t index;
  size_t bytes;
  size_t blkLength = gcry_cipher_get_algo_blklen(GCRY_CIPHER);

  if(!fp) return 1;

  // setup encryption
  if(gcry_cipher_open(&hd, GCRY_CIPHER, GCRY_CIPHER_MODE_CBC,
     ↪ 0)) return 1;
  if(gcry_cipher_setkey(hd, key,
     ↪ gcry_cipher_get_algo_keylen(GCRY_CIPHER))) return 1;
  if(gcry_cipher_setiv(hd, iv, blkLength)) return 1;

  for(;;) {
    start = clock(); // begin time measure
```

```
    memset(buf, 0, BLOCK_SIZE);
    bytes = fread(buf, sizeof(char), BLOCK_SIZE, fp);
    if(gcry_cipher_encrypt(hd, buf, BLOCK_SIZE, NULL, 0))
       ↪ return 1;
    for(index = 0; index<bytes; index++)
      fprintf(stdout, "%02X", (unsigned char)buf[index]);
    fprintf(stdout, "\n");
    fprintf(stderr, "Current encryption speed: %lu seconds per
       ↪ byte\n", (clock() - start) / CLOCKS_PER_SEC / bytes);
    if(bytes < BLOCK_SIZE && feof(fp)) break;
  }
  fprintf(stdout, "\n");
  gcry_cipher_close(hd);
  free(buf);
  return 0;
}
```

This is a simple program, that uses libgcrypt to encrypt the content of the file `secret` in the current directory and output the result to stdout. Additionally, we measure performance and write debug output to stderr after every 16 encrypted bytes. Before you can build it, you have to make sure, libgcrypt-dev is installed:

```
$ sudo aptitude install libgcrypt-dev
$ gcc -o hsar hsar.c $(libgcrypt-config --cflags --libs)
$ echo 'Soft kitty, Warm kitty, Little ball of fur. Happy
   ↪ kitty, Sleepy kitty, Purr, purr, purr' >secret
$ ./hsar
5A08EFAAC56911D407F85AC1D959CA71
Current encryption speed: 0 seconds per byte
9E8F67D7934C73B90FDD4DB4EA5E8F72
Current encryption speed: 0 seconds per byte
C3A19055A5551682E5E4031701302179
Current encryption speed: 0 seconds per byte
41B28B78D63A9FD9AE3CA05CFC5F5F0F
Current encryption speed: 0 seconds per byte
84BB71ECC01D5878D3A02E17F04F4305
Current encryption speed: 0 seconds per byte
```

```
98FD0286246E81A2
Current encryption speed: 0 seconds per byte
```

We notice, that apart from the encryption being very fast, the program crashes if we truncate the input a little bit:

```
$ echo 'Soft kitty, Warm kitty, Little ball of fur. Happy
   ↪ kitty, Sleepy' >secret
$ ./hsar
5A08EFAAC56911D407F85AC1D959CA71
Current encryption speed: 0 seconds per byte
9E8F67D7934C73B90FDD4DB4EA5E8F72
Current encryption speed: 0 seconds per byte
C3A19055A5551682E5E4031701302179
Current encryption speed: 0 seconds per byte
DCFE4636C9A1D7B6763C18D135CF680F
Current encryption speed: 0 seconds per byte
Floating point exception
```

Next you will use Monkey to find the bug.

6.1.4 Static Code Analysis

First you have to generate an expression database with Cparser and Seaspider so that Pathologist knows which expressions to evaluate. To do this, you have to compile your code with Cparser and the mandatory `CFLAGS`. If you want to use Cparser on a different project that uses a build system like autoconf, just `export CC=$(which cparser)` before the `./configure` step as well.

```
$ export CFLAGS="-m32 --seaspider"
$ cparser -o hsar.o -c hsar.c $(libgcrypt-config --cflags)
   ↪ $CFLAGS
```

Now you should have two new files in the directory: `hsar.sea` and `hsar.seahtml`. The `.sea` file is a text file containing all expressions from `hsar.c` and `hsar.seahtml` is a syntax highlighted copy of `hsar.c` formated as HTML, which can be used later in Scrutinizer.

Be aware that Cparser often spits out compiler warnings that you do not see with other compilers. In rare cases, you might see an error with Cparser although gcc compiles your code just fine. In this case you will have to fix the compiler error to get

all expressions from the corresponding source file. However, note that we will not use the generated binary as it is currently not compatible with GDB. A bigger issue is that Cparser only works for x86, and while you can cross-compile on AMD64 to x86, your dependencies might then not be available as 32-bit libraries on the AMD64 system. Thus, you might find it easier to use a 32 bit machine to generate the expression database. Again, these restrictions only matter for building the expression database as you will not deploy the resulting binary; even Cparser generating incorrect code is not an issue for Monkey. Still, if you have problems with Cparser, you might want to contact the Cparser developers.

Now you have to use Seaspider to convert the `hsar.sea` file into an sqlite database. Since you do not want the super secret encryption key and initialization vector or the buffer `buf` (which contains part of the secret message) to be included in the reports, you have to write an ignore file first. This file will tell Pathologist which expressions must not be included in reports as they are private. You can read more about excluding expressions in the man page of Seaspider. You can generate an 'ignores' file for the given example and then run Seaspider using the following commands:

```
$ echo 'hsar\.c:main:key' > ignores
$ echo 'hsar\.c:main:iv' >> ignores
$ echo 'hsar\.c:main:buf' >> ignores
$ seaspider -i ignores
```

After this command you should have your `hsar.sqlite` file ready. Note, that Seaspider per default scans all subdirectories for `.sea` files recursively so you can just run it from your source directory after running Cparser.

6.1.5 Report Generation

To get a report from the crashing program, you will run it within Pathologist. Since Pathologist uses GDB internally, we first have to build our binary again with gcc and this time ensure that the `-g` (debugging information) flag is set so that debug symbols are included:

```
$ gcc -g -o hsar hsar.c $(libgcrypt-config --cflags --libs)
```

Afterwards, you can run your application with Pathologist using the following arguments:

```
$ pathologist -d hsar.sqlite -o hsar.json -p20 -- hsar
```

The arguments are pretty straighforward:

- `-d hsar.sqlite` specifies the path to the expression database
- `-o hsar.json` specifies the path to the report to be generated
- `-p20` specifies the maximum stack depth to traverse
- `--` separates arguments to Pathologist from the client binary and arguments
- `hsar` is the client binary to debug

After the binary name (here `hsar`), you could specify arbitrary additional command-line arguments for the application.

If everything has been setup correctly, Pathologist should be able to run GDB, catch the SIGFPE signal, evaluate the expressions and generate the report file. If you want, you can view the report in any text editor (with JavaScript syntax highlighting) or in Scrutinizer. Naturally, in a real-world deployment, you might want to add logic to send the report back to the developer (via e-mail or HTTP), possibly after confirmation from the user.

6.2 Example Report

```
1  {
2    "category": "assertion failure",
3    "sysinfo": {
4      "time": "2013-07-03 17:17:49+02:00",
5      "os": "Linux",
6      "os_release": "3.2.0-42-generic",
7      "hardware": "x86_64",
8      "gcc": "gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3",
9      "dependencies of /bin/ls": {
10       "libselinux.so.1": "/lib/x86_64-linux-gnu/libselinux.so.1",
11       "librt.so.1": "/lib/x86_64-linux-gnu/librt.so.1",
12       "libacl.so.1": "/lib/x86_64-linux-gnu/libacl.so.1",
13       "libc.so.6": "/lib/x86_64-linux-gnu/libc.so.6",
14       "libdl.so.2": "/lib/x86_64-linux-gnu/libdl.so.2",
15       "libpthread.so.0": "/lib/x86_64-linux-gnu/libpthread.so.0",
16       "libattr.so.1": "/lib/x86_64-linux-gnu/libattr.so.1"
17     }
```

```
18  },
19  "history": [{
20     "trace": [{
21        "function": "get_net_param_from_list",
22        "line": 2192,
23        "file": "src/or/networkstatus.c",
24        "expressions": {
25          "net_params": "0x555555945150",
26          "param_name": "0x5555556bea90 \"GuardLifetime\"",
27          "default_val": 2592000,
28          "min_val": 5184000,
29          "max_val": 157766400,
30          "__builtin_expect": null,
31          "(max_val > min_val)": 1,
32          "(!(max_val > min_val))": 0,
33          "!(!(max_val > min_val))": 1,
34          "!!(!(max_val > min_val))": 0,
35          "log_fn_": "{void (int, log_domain_mask_t, const char
               ↪ *, const char *, ...)} 0x555555667dd0 <log_fn_>",
36          "1u": 1,
37          "(1u << 12)": 4096,
38          "__PRETTY_FUNCTION__": "[24]",
39          "(\"src/or/networkstatus.c\")": "[23]",
40          "__func__": "[24]",
41          "fprintf": "{<text variable, no debug info>}
               ↪ 0x7ffff6abf7b0 <__fprintf>",
42          "stderr": "0x7ffff6e25180",
43          "abort": "{<text variable, no debug info>}
               ↪ 0x7ffff6aa5a10 <__GI_abort>",
44          "(min_val <= default_val)": 0,
45          "(!(min_val <= default_val))": 1,
46          "!(!(min_val <= default_val))": 0,
47          "!!(!(min_val <= default_val))": 1
48        }
49      }, {
50        "function": "guards_get_lifetime",
```

```
51          "line": 472,
52          "file": "src/or/entrynodes.c",
53          "expressions": {
54            "options": "<optimized out>",
55            "options->GuardLifetime": "",
56            "options->GuardLifetime >= 1": "",
57            "networkstatus_get_param": "{int32_t (const
                ↪ networkstatus_t *, const char *, int32_t,
                ↪ int32_t, int32_t)} 0x555555571a20
                ↪ <networkstatus_get_param>",
58            "((void*)0)": "0x0"
59          }
60        }, {
61          "function": "remove_obsolete_entry_guards",
62          "line": 487,
63          "file": "src/or/entrynodes.c",
64          "expressions": {
65            "now": 1372864669
66          }
67        }, {
68          "function": "entry_guards_compute_status",
69          "line": 613,
70          "file": "src/or/entrynodes.c",
71          "expressions": {
72            "entry_guards": "0x55555591b560",
73            "!entry_guards": 0,
74            "options": "0x5555559147d0",
75            "options->EntryNodes": "0x0",
76            "entry_nodes_should_be_added": "{void (void)}
                ↪ 0x555555651790 <entry_nodes_should_be_added>",
77            "reasons": "0x555555ea2b80",
78            "digestmap_new": "{digestmap_t *(void)}
                ↪ 0x555555663bf0 <digestmap_new>",
79            "now": 1372864669,
80            "changed": 1,
```

```
81          "remove_obsolete_entry_guards": "{int (time_t)}
               ↪ 0x5555556524e0 <remove_obsolete_entry_guards>"
82        }
83       }, {
84        "function": "directory_info_has_arrived",
85        "line": 984,
86        "file": "src/or/main.c",
87        "expressions": {
88          "router_have_minimum_dir_info": "{int (void)}
               ↪ 0x555555576c80 <router_have_minimum_dir_info>",
89          "directory_fetches_from_authorities": "{int (const
               ↪ or_options_t *)} 0x555555634890
               ↪ <directory_fetches_from_authorities>",
90          "options": "0x5555559147d0",
91          "entry_guards_compute_status": "{void (const
               ↪ or_options_t *, time_t)} 0x555555654770
               ↪ <entry_guards_compute_status>",
92          "now": 1372864669
93        }
94       }, {
95        "function": "do_main_loop",
96        "line": 1927,
97        "file": "src/or/main.c",
98        "expressions": {
99          "dns_init": "{int (void)} 0x55555564a8c0 <dns_init>",
100         "client_identity_key_is_set": "{int (void)}
               ↪ 0x55555559de50 <client_identity_key_is_set>",
101         "connection_bucket_init": "{void (void)}
               ↪ 0x555555607370 <connection_bucket_init>",
102         "stats_prev_global_read_bucket": 1073741824,
103         "global_read_bucket": 1073741824,
104         "stats_prev_global_write_bucket": 1073741824,
105         "global_write_bucket": 1073741824,
106         "control_event_bootstrap": "{void
               ↪ (bootstrap_status_t, int)} 0x555555625cf0
               ↪ <control_event_bootstrap>",
```

```
107         "trusted_dirs_reload_certs": "{int (void)}
               ↪ 0x5555555ade30 <trusted_dirs_reload_certs>",
108         "router_reload_v2_networkstatus": "{int (void)}
               ↪ 0x555555571430 <router_reload_v2_networkstatus>",
109         "router_reload_consensus_networkstatus": "{int
               ↪ (void)} 0x555555572c10
               ↪ <router_reload_consensus_networkstatus>",
110         "router_reload_router_list": "{int (void)}
               ↪ 0x5555555b0fa0 <router_reload_router_list>",
111         "now": "<optimized out>",
112         "time": "{void *(void)} 0x7ffff6b1c040 <time>",
113         "((void*)0)": "0x0",
114         "directory_info_has_arrived": "{void (time_t, int)}
               ↪ 0x5555555693f0 <directory_info_has_arrived>"
115       }
116     }, {
117       "function": "tor_main",
118       "line": 2696,
119       "file": "src/or/main.c",
120       "expressions": {
121         "update_approx_time": "{void (time_t)} 0x55555566ecc0
               ↪ <update_approx_time>",
122         "time": "{void *(void)} 0x7ffff6b1c040 <time>",
123         "((void*)0)": "0x0",
124         "tor_threads_init": "{void (void)} 0x55555565fed0
               ↪ <tor_threads_init>",
125         "init_logging": "{void (void)} 0x555555667220
               ↪ <init_logging>",
126         "tor_init": "{int (int, char **)} 0x555555569dc0
               ↪ <tor_init>",
127         "argc": "<optimized out>",
128         "argv": "0x7fffffffd5e8",
129         "get_options": "{const or_options_t *(void)}
               ↪ 0x5555555facc0 <get_options>"
130       }
131     }, {
```

```
132         "function": "__libc_start_main",
133         "line": 226,
134         "file": "libc-start.c",
135         "expressions": null
136       }, {
137         "function": "_start",
138         "line": 0,
139         "file": null,
140         "expressions": null
141       }],
142      "globals": null
143    }]
144 }
```

6.3 GDB Output Tags

In the GDB output some additional information can occur in tags. This list describes the tags and how to interpret them in the context of Pathologist.

- <synthetic pointer>
  Synthetic pointers are a DWARF extension typically used when the compiler supports SRA. GCC emits this in some cases. No occurence seen yet, we can probably ignore it.

- <repeats %u times>
  Can be omitted with 'set print elements 0'.

- <invalid address>
  Seems to be only used for Pascal and C++ source. We ignore this tag.

- <error reading variable>
  Can occur in various unknown circumstances. Probably no bug related information can be obtained from this tag, so we ignore it.

- <address of value unknown>
  Seems to be for errorhandling in GDB.

- <internal function %s>
  Pathologist does not expose internal functions to GDB.

- <incomplete sequence %WHATEV>
  Seems to only occur, when GDB interprets something as a wchar array and finds a "half" wchar. We probably should ignore it.

- <unavailable>
  Seems only to be relevant for advanced GDB features which are not used by Pathologist.

- <Error reading address %HEX?: %s>
  Should be added to the report, possibly with /proc/PID/maps info for the address.

- <Address %HEX out of bounds>
  Should be added to the report, possibly with /proc/PID/maps info for the address.

- <invalid float value>
  Invalid floating point number, should be added to the report. Could also indicate a serious problem like stack corruption.

- <incomplete type>
  Should be added to the report. Occurs if, e.g., type is a struct without members.

- <optimized out>
  Should be added to the report, so developers can decide to make the variable volatile or try with "gcc -O0".

7 Bibliography

[1] Alessandro Orso Chris Parnin. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[2] Safey A. Halim. Monkey: Automated debugging of deployed distributed systems. Master's thesis, TUM, July 2012.

[3] mmentovai@gmail.com. Google-breakpad cliend design. `http://code.google.com/p/google-breakpad/wiki/ClientDesign`. last visited on 2012-10-14.

[4] mmentovai@gmail.com. Google-breakpad processor design. `http://code.google.com/p/google-breakpad/wiki/ProcessorDesign`. last visited on 2013-05-12.

[5] Mishra; Nayak; Rath; Swain. Far efficient k-means clustering algorithm. In *ICACCI*, 2012.