Oops! Where Did That Code Snippet Come From?

Lisong Guo, Julia Lawall, Gilles Muller

To cite this version:

Lisong Guo, Julia Lawall, Gilles Muller. Oops! Where Did That Code Snippet Come From?. MSR 2014 - 11th Working Conference on Mining Software Repositories, May 2014, Hyderabad, India. pp.52-61, 10.1145/2597073.2597094 . hal-01080397

HAL Id: hal-01080397

https://hal.inria.fr/hal-01080397

Submitted on 5 Nov 2014

Oops! Where Did That Code Snippet Come From?

Lisong Guo, Julia Lawall, Gilles Muller
Inria/LIP6-Regal, Sorbonne Universités, UPMC Univ Paris 06, Paris, France
{Lisong.Guo, Julia.Lawall, Gilles.Muller}@lip6.fr

ABSTRACT

A kernel oops is an error report that logs the status of the Linux kernel at the time of a crash. Such a report can provide valuable first-hand information for a Linux kernel maintainer to conduct postmortem debugging. Recently, a repository has been created that systematically collects kernel oopses from Linux users. However, debugging based on only the information in a kernel oops is difficult. We consider the initial problem of finding the offending line, *i.e.*, the line of source code that incurs the crash. For this, we propose a novel algorithm based on approximate sequence matching, as used in bioinformatics, to automatically pinpoint the offending line based on information about nearby machine-code instructions, as found in a kernel oops. Our algorithm achieves 92% accuracy compared to 26% for the traditional approach of using only the oops instruction pointer.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms

Keywords

Linux kernel, oops, debugging, sequence alignment

1. INTRODUCTION

The Linux kernel is used today in environments ranging from embedded systems to servers. While the Linux kernel is widely regarded as being stable for ordinary use, it nevertheless still contains bugs [17]. Indeed, the Linux kernel offers a huge variety of services (devices drivers, file systems, etc.), and a huge range of configuration options [20], making it essentially impossible to exhaustively test. Thus, Linux developers must rely on bug reports to understand the full scope of possible problems with their code.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597094

Recently, a repository of Linux kernel crash reports, referred to as *kernel oopses*, has become available [2]. This repository systematically collects kernel oopses from any user whose Linux distribution has the oops reporting facility enabled. As such, the repository has the potential to give the kernel developer a much more systematic view of the commonly encountered problems in his code. An oops report, however, provides only an instantaneous image of the kernel state at the time of the crash, expressed primarily at the level of the machine code of the running kernel. This makes kernel oopses difficult to interpret, drastically limiting the practical benefit of the repository. A proposal to extend the Linux kernel to make it possible to include line number information was recently rejected, as it would incur a too great (albeit small) space overhead in a running kernel [12].

A fundamental part of debugging any software crash is to identify the *offending line*, *i.e.*, the line of the source code at which the crash occurs. To illustrate the importance of knowing the offending line and the difficulty of obtaining it from a kernel oops, even in the ideal case when it is possible to interact with the user who encountered the crash, we describe a real story from the Linux kernel bugzilla [4]. A user named Jochen encountered frequent crashes on his XFS file system and filed a bug report that included a kernel oops. A developer named Dave took charge of the report, but he did not understand the issue. As Dave did not have access to the compiled kernel that Jochen had used, he instructed Jochen to go through a series of procedures on the victim machine in order to produce the offending line information. In all, it took 16 comment exchanges before another user, Katharine, who had the same problem, managed to obtain the offending line. Dave then quickly figured out the source of the bug and provided a corresponding patch.

In contrast to the bugzilla case, when a developer finds an oops in the kernel oops repository that may be relevant to his code, there is no associated user that the developer can contact. This compounds the difficulty of finding the offending line. Several options remain. If the developer can deduce the exact version and distribution from the oops, then he may be able to retrieve a precompiled version of that kernel with debugging information from that distribution, if such a kernel is available. Alternatively, the developer can compile the kernel file containing the crashing function from the same Linux version, to try to recreate the victim kernel's binary code. If either of these is successful, the developer can load the kernel code into the debugger gdb and request the source code line corresponding to the instruction pointer mentioned in the oops. A second option is to disassemble the small snippet of binary code surrounding the crash site, that is found in oopses corresponding to the more severe types of bugs, and then match the result against the assembly code of the crashing function. A third option is to analyze the semantics of the instructions found in the disassembled code snippet, to relate this semantics to the source code.

In this paper, we argue that the second approach, of matching the disassembled code snippet to the assembly code of a locally compiled kernel, is the most promising. The first approach, using the instruction pointer, is lightweight but brittle, especially in the common case where it is not possible to retrieve the distribution's original debugging kernel. In contrast, the code snippet used in the second approach provides more context information, and is thus more tolerant to variations in locally generated code. The third approach, relating the semantics of the disassembled code snippet directly to the source code, is very difficult, due to the different levels of abstraction involved. The second approach nevertheless has the disadvantage that the matching process is tedious and error prone, especially for the many large kernel functions. To make the second approach practical, we propose to automate the matching process, based on a variant of *approximate string matching*, as used in bioinformatics [22].

We have evaluated our approach on 100 randomly selected examples from the kernel oops repository and the Linux kernel bugzilla. Our approach achieves 92% accuracy, compared to the 26% accuracy obtained when relying solely on the instruction pointer. It is thus effective and efficient, having complexity linear in the size of the crashing function, and, as illustrated by the bugzilla example, produces information that can ease the burden on a Linux kernel maintainer in debugging a kernel oops.

The contributions of this paper are as follows:

- We quantify the difficulty of finding the offending line in an oops generated by an older kernel.
- We propose and formalize an algorithm using approximate sequence matching for identifying the line of source code that led to the generation of an oops.
- We show that the complexity of our algorithm is linear in the size of the crashing function.
- We show that our approach is effective, achieving an accuracy of 92% on 100 randomly selected examples.

The rest of the paper is organized as follows. Section 2 gives some background, including the content of a kernel oops and an illustration of the standard approach to obtaining the offending line. Section 3 presents the outline of our matching algorithm. Section 4 instantiates this algorithm with some domain-specific design choices. We then evaluate our solution in Section 5 and discuss the validities of our solution in Section 6. We review some related work in Section 7. Finally, we conclude in Section 8.

2. BACKGROUND

We describe the structure of a kernel oops and existing approaches to finding the offending line. We then present some statistics that illustrate the difficulty of the latter process.

2.1 Key Attributes of a Kernel Oops

A kernel oops [23] consists of the information logged by the Linux kernel when a crash or warning occurs. It is composed mostly of *attribute*: *value* pairs, in which each *value* records a specific piece of information about certain attribute of the system. The format and content of a kernel oops depend somewhat on the cause of the oops and the architecture of the victim machine. We focus on the x86 architecture (both 32 bit and 64 bit), which currently represents almost all of the oopses found in the kernel oops repository. Figure 1 shows the XFS kernel oops discussed previously. Some values are omitted (marked with ...) for brevity. Some key attributes ($\boxed{\text{boxed}}$) are:

Oops description. A kernel oops begins with a brief description of the cause of the oops. In our example, the oops was caused by NULL-pointer dereference (line 1).

Error site. The *IP* (Instruction Pointer, line 2) field indicates the name (`xfs_da_do_buf`) of the function being executed at the time of the oops, *i.e.*, the crashing function, the binary code offset (`0x4da`) at which the oops occurred, and the binary code size (`0x5e1`) of the crashing function.

Kernel version. A string (`2.6.36-gentoo-r5`) following the value of the attribute *Tainted* (line 5) indicates the Linux kernel version (`2.6.36`, maintained between October 2010 and February 2011) from which the kernel oops was generated. The string may also indicate the distribution of the Linux kernel, here Gentoo.

Code snippet. Some kernel oopses, for the more severe faults, contain a binary code snippet (line 13) indicating the 64 bytes of binary code surrounding the error site (43 bytes before the trapping instruction and 20 after), regardless of instruction boundaries. Within this snippet, the first byte of the instruction that causes the crash is marked by <>. We refer to this instruction as the *trapping instruction.*

```
1  BUG: unable to handle kernel
         NULL pointer dereference at 00000008
2  IP: [<c10fb360>] xfs_da_do_buf+0x4da/0x5e1
3  Oops: 0000 [#1] PREEMPT SMP
4  Modules linked in: nfs lockd nfs_acl sunrpc w83627hf ...
5  Pid: 2725, comm: rm Tainted: P 2.6.36-gentoo-r5 #10
6  EIP: 0060:[<c10fb360>] EFLAGS: 00010246 CPU: 0
7  EAX: 00000001 EBX: f60da400 ECX: fbd5a730 EDX: 00000000 ...
8  Stack: 0012c096 00000000 f64cabc0 f64ca5dc d853fb40 ...
9  Call Trace:
10  [<c1020717>] ? get_parent_ip+0xb/0x31
11  [<c102086e>] ? sub_preempt_count+0x7c/0x89
12  [<c111f85e>] ? xfs_remove+0x1b3/0x2e0 ...
13 Code:  f0 00 c7 45 b8 00 00 00 00 74 13 8b 4d 18
    8d 55 f0 b8 01 00 00 00 e8 06 fa ff ff 89 45 b8 83 7d 14
    01 0f 85 82 00 00 00 8b 55 b8 <8b>  4a 08 8b 51 08 8b 01
    0f c8 86 f2 0f b7 d2 81 fa ee fb 00 00
14 ---[ end trace 118398ff1b25f91d ]---
```

Figure 1: XFS kernel oops

2.2 Pinpointing the Offending Line

We now consider in more detail the three options for pinpointing the offending line that were proposed in the introduction, in terms of the example in Figure 1. We place ourselves in the situation where a kernel with debugging information corresponding to the offending kernel is not available, as finding such debugging kernels is difficult for versions that do not correspond to a current release of the given distribution, and even if possible requires somewhat obscure distribution-specific knowledge.

Gdb. To use gdb on the instruction pointer found in the kernel oops, we first need to obtain binary code with debugging information for the file containing the crashing function. For this, we download the kernel version corresponding to the version (2.6.36) listed in the kernel oops from the mainline kernel repository `kernel.org`. We then create a default configuration file, using the command `make ARCH=i386 defconfig` because the oops contains 32 bit addresses. In the configuration file, to obtain debugging information, we then manually select the option `CONFIG_DEBUG_INFO` and unselect the option `DEBUG_INFO_REDUCED`. We then use the local version of gcc (in our case, gcc 4.7.2 (Debian 4.7.2-5)) to compile the file containing the crashing function, using the command `make ARCH=i386 fs/xfs/xfs_da_btree.o`. Finally, we load the resulting object file into gdb and obtain the offending line with the command `list *(xfs_da_do_buf+0x4da)`.

Having followed this procedure, we obtain a line that is five lines away from the one indicated in the bugzilla discussion. There are furthermore several lines at which a NULL pointer dereference could occur between the line identified by gdb and the actual offending line. This approach is clearly too brittle for practical use.

Assembly code matching. For this approach, we first obtain a locally compiled debugging kernel, as above. We then use the Linux kernel script `decodecode`, with the environment parameter `AFLAGS=--32 | --64` indicating the architecture, to disassemble the code snippet. Next, we use gdb with the command `disassemble xfs_da_do_buf` to disassemble the crashing function. We must then manually match the two sequences of assembly code to find the offset of the instruction in the locally compiled crashing function that most probably matches the trapping instruction in the code snippet. This offset can then be used with gdb as in the previous case, to get the offending line.

In the case of the oops shown in Figure 1 the locally compiled crashing function consists of 459 instructions, as compared to the 19 instructions in the code snippet. The large size of the crashing function makes the matching process tedious and error prone. Indeed, an exact match is not likely, due to the use of different compilers or compiling options, which causes differences in both the choice of instructions and the order in which they appear. These problems can be somewhat alleviated if the instruction pointer indicates that the trapping instruction is very near the beginning or end of the function. Our example, however, does not have this property.

At the other end of the spectrum, we consider an oops derived from the much smaller function shown in Figure 2. Figure 3 shows the correspondence between the code snippet in the oops and the assembly code obtained from locally compiling the crashing function. By comparing the sequence of opcodes, we can see that the trapping instruction at offset `<+8>` in the code snippet corresponds to the instruction at offset `<+13>` in the locally compiled crashing function. The debugging information then indicates that the instruction at offset `<+13>` of the crashing function originates from the statement at line 4 in Figure 2, which is the offending line. During the matching process, some gaps, marked by `---`, are required, and we can see that the compilers have taken different decisions in compiling the conditional test, marked with arrows at offsets `<+15>` and `<+18>`, respectively. These issues complicate the matching even for a small example.

```
1 int elv_may_queue(struct request_queue *q, int rw)
2 {
3       struct elevator_queue *e = q->elevator;
4       if (e->ops->elevator_may_queue_fn)
5             return e->ops->elevator_may_queue_fn(q, rw);
6       return ELV_MQUEUE_MAY;
7 }
```

Figure 2: The source code of the crashing function

code snippet	crashing function
`<+0>  push %ebx`	`<+0>  push %ebp`
`<+1>  mov %eax,%ebx`	`<+1>  mov %esp,%ebp`
`− − −`	`<+3>  call 0x1414 (<+4>)`
`<+3>  mov 0xc(%eax),%eax`	`<+8>  mov 0xc(%eax),%ecx`
`<+6>  mov (%eax),%eax`	`<+11> mov (%ecx),%ecx`
`<+8>  mov 0x38(%eax),%ecx`	`<+13> mov 0x38(%ecx),%ecx`
`<+11> xor %eax,%eax`	`− − −`
`<+13> test %ecx,%ecx`	`<+16> test %ecx,%ecx`
`<+15> je 0x38 <+21>`	`<+18> je   0x1428 <+24>`
`<+17> mov %ebx,%eax`	`− − −`
`<+19> call *%ecx`	`<+20> call  *%ecx`
`<+21> pop %ebx`	`<+22> pop   %ebp`
`<+22> ret`	`<+23> ret`
`− − −`	`<+24> xor   %eax,%eax`
`− − −`	`<+26> pop   %ebp`
`− − −`	`<+27> nop`
`<+23> lea 0x4c(%edx),%eax`	`<+28> lea 0x0(%esi,%eiz,1),%esi`
`− − −`	`<+32> ret`

Figure 3: An example of assembly code matching

Semantics matching. For this approach, we use `decodecode` as above to obtain the assembly code of the code snippet and download the kernel version mentioned in the oops, but do not compile it. For simplicity, we illustrate this approach using only our second example (Figures 2 and 3). We reason as follows: I) The trapping instruction is a memory access operation, due to the operand `0x38(%eax)`, which implies that the offending line involves a pointer dereference. II) Right above the trapping instruction, there is a similar instruction that operates on the same register as the trapping instruction. Thus, the offending line likely contains two successive pointer-dereferences, *i.e.*, it has the form $a \to b \to c$, for some expression a, and some fields b and c. III) Following the trapping instruction, there is a branching test which should correspond to a conditional test in the source code. IV) Furthermore, the result of the trapping instruction is stored in the register `%ecx`, which is used later at offset `<+19>` as a function pointer (`call *%ecx`). Thus, the offending line should evaluate to a function pointer for a following invocation. Line 4 in the crashing function is the only one that satisfies all of the above properties, and is indeed the offending line.

Although successful in this case, this approach is clearly not systematic enough for frequent use, particularly for large or complex crashing functions, such as the one associated with Figure 1. Nevertheless, we use this approach, combined with some manual matching of the assembly code to validate our results obtained in Section 5.

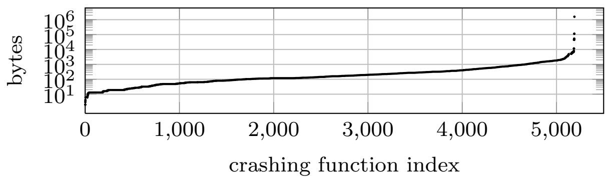

Figure 4: Function size for unique oopses containing both a code snippet and function name

2.3 Properties of Oopsing Functions

To better understand the problems confronting a kernel developer, we have studied various properties of the oopses collected during the first 8 months of the existence of the kernel oops repository.

Data set. Between September 1, 2012 and May 1, 2013, the kernel oops repository received 187,342 kernel oopses. Many are warnings, and thus only 22,316 contain a code snippet. Our approach requires the presence of both a code snippet and a function name. 3263 of the code snippets contain only "Bad EIP value" or "Bad RIP value" indicating an invalid instruction pointer. Of the remaining 19,053 oopses, 897 do not give a name for the crashing function, only an address, again indicating an invalid instruction pointer. Finally, the kernel oops repository contains many duplicate reports [8]. Of the oopses containing both a function name and a code snippet, 5192 have a unique combination of function name, crashing instruction offset, function size, and code snippet. We consider only these oopses in the rest of this section.

Size of the crashing function. As the size of the crashing function increases, it becomes more difficult to scan through its assembly code to find instructions that look similar to the disassembled code snippet. Figure 4 shows the sizes in bytes of the crashing functions indicated in our oopses, ordered from smallest to largest.[1] This figure shows one point per oops, even though multiple oopses may refer to the same function. 20% of the unique oopses involve functions containing more than 500 bytes, making manual scanning for a close match to a 64-byte code snippet impractical. Three oopses refer to a function that contains over a million bytes. This is a large memory region used by a kernel module added by VirtualBox, and is not part of the mainline kernel.[2]

Compiler effects. A major source of variation in the sequence of instructions generated by compiling a kernel is the version of the compiler used. To measure this effect, we consider three compilers: gcc 4.6.1 (Ubuntu/Linaro 4.6.1-9ubuntu3), gcc 4.7.2 (Debian 4.7.2-5), and gcc 4.8.1 (Ubuntu/Linaro 4.8.1-10ubuntu9). For simplicity, we test these compilers on a single version, Linux 3.7, released in 2012. For this kernel, we generate a configuration file using `make ARCH=x86_64 defconfig`. Linux 3.7 represents 12% of our considered oopses, and is the second most common version mentioned. The most common version, representing 57% of the oopses, is Linux 2.6.32, first released in 2009. Our compilers, however, generate many warnings on the Linux 2.6.32 code, and we want to avoid comparing invalid code.

[1]We use the number of bytes indicated in the oops; the number of instructions is not available.

[2]https://forums.virtualbox.org/viewtopic.php?f=7&t=30037

Table 1: Compiler comparison.

	1 gcc version						2 gcc versions		
	4.6.1 → 4.7.2			4.7.2 → 4.8.1			4.6.1 → 4.8.1		
	instr diff	sz diff	# fns	instr diff	sz diff	# fns	instr diff	sz diff	# fns
small	14%	8%	353	14%	4%	350	23%	11%	354
medium	17%	4%	404	21%	4%	402	28%	6%	397
large	32%	3%	102	32%	4%	102	40%	5%	103

Instr diff is the rate of difference in the instructions. Sz diff is the rate of difference in the function sizes. #fns is the number of functions that are small, medium, or large, respectively.

Our single tested kernel version and configuration may not cover all the crashing functions of our 5192 oopses. There are 1557 such functions, of which 25% are either not defined in Linux 3.7, are only defined in architecture-specific code (`arch` directory) for an architecture other than x86, or are only defined in header files, which do not generate an independent object file in which we can find the binary code definition. Finally, for 18% of the 1557 functions, the defining file is not included in our kernel configuration, and for 3% of the 1557 functions, at least one compiler does not generate code, perhaps due to inlining. This leaves just over 850 distinct crashing functions whose assembly code we can compare.

For each compilation result, we first obtain the assembly code, using the command `objdump -disassemble`, and then rewrite the output to abstract away all information except the opcode, any constants, and any constant offsets used in indirect references. This eliminates the effect of compiler choices such as register names and branch offsets, and focuses on information that is likely to stand out as the developer scans the code. To estimate the difference between two compilation results we apply the unix tool `diff` to these abstracted versions, and then divide the sum of the number of differences by twice the number of instructions found in the smaller compiled function, to normalize the results. As a baseline, we also compute the difference in size between the smaller and larger compiled functions as a percentage of the size of the smaller compiled function. Finally, we distinguish between small functions, for which the smaller compiled function contains 50 instructions or less, medium functions, for which the smaller compiled function contains 250 instructions or less, and large functions, for which the smaller compiled function contains more than 250 instructions. The number of functions in each category varies slightly depending on which compilers are being compared.

The results are shown in Table 1. In each case, the comparison on instruction opcodes and operands shows a much greater rate of change than the comparison on code size. Thus, many changes must be within the shared part of the code; there is not *e.g.*, simply a new sequence of code added at the end of the function. Furthermore, we see that the rate of changes increases as the function size increases and as the distance between the compiler versions increases. All these variations make manual matching complex.

3. AUTOMATING ASSEMBLY CODE MATCHING

Our goal is to automate the matching of the disassembled code snippet against the disassembled locally compiled crashing function, as previously illustrated in Figure 3. We take inspiration from *approximate sequence matching*, used in bioinformatics [22]. We begin with some definitions, and then propose a matching algorithm that focuses on each possible counterpart to the trapping instruction and its surrounding context. This matching algorithm uses an algorithm for matching a code sequence against a prefix of another code sequence, which we present next. Finally, we consider the complexity of the complete approach. Some design decisions related to the particular properties of assembly language instructions are deferred to Section 4.

3.1 Definitions

We refer to disassembled code snippet as the *code snippet*, C, and the disassembled locally compiled crashing function as the *local function*, L. We first define some properties of sequences, and then describe sequence matching:

Definition 1. *For a sequence S, let $S[i]$ denote the element at the position i, and $|S|$ denote the length of S. We also define a special element called a* gap, *denoted $\perp$.*

Definition 2. *For a sequence S of length n and any $1 \leq i \leq j \leq n$, $\widehat{S} = S[i...j]$ is a* substring *of S and $\widehat{S} = S[1...j]$ is a* prefix *of S.*

Definition 3. *For sequences S_1 and S_2, we define an* anchored alignment *of S_1 with S_2, written* $\mathrm{Align}(S_1, S_2)$*, as any (S_1', S_2') where: I). S_1' and S_2' are sequences that may contain gaps. II). $|S_1'| = |S_2'|$. III). Removing gaps from S_1' leaves S_1. IV). Removing gaps from S_2' leaves a* prefix *of S_2.*

Anchored alignment is a variant of the *global alignment* and *local alignment* used in bioinformatics [22]. Global alignment matches two complete sequences, while local alignment matches subsequences of those sequences. Anchored alignment matches a complete sequence against a prefix of another sequence, with gaps allowed. Figure 5 shows an example.

$$\begin{array}{ccccccccc} S_1[1] & \perp & S_1[2] & S_1[3] & S_1[4] & \perp & \perp & S_1[5] & \\ | & | & | & | & | & | & | & | & \\ S_2[1] & S_2[2] & S_2[3] & S_2[4] & \perp & S_2[5] & S_2[6] & S_2[7] & \ldots \end{array}$$

Figure 5: *Align*(S_1, S_2), where $|S_1| = 5$ and $|S_2| \geq 7$

In general, for finding the offending line, it is not sufficient to use any match; instead, we want one that is considered to be best according to some scoring function. Indeed, the challenge in performing alignment is to determine where to place gaps to obtain the highest possible score.

Definition 4. *For elements s_1 and s_2, let* $\mathrm{score}(s_1, s_2)$ *be a measure of the similarity between s_1 and s_2. The precise definition of* score *is orthogonal to the problem of matching, and is deferred to the next section. Nevertheless, we require that a match with a gap have a negative score, to make it unfavorable to match a gap with itself.*

Definition 5. *For sequences S_1' and S_2' of length n, which may contain gaps, we define the* similarity *of S_1' with S_2', as* $\mathrm{Sim}(S_1', S_2') = \sum_{i}^{n} \mathrm{score}(S_1'[i], S_2'[i])$.

Definition 6. *For sequences S_1 and S_2, let Θ be the set of all possible anchored alignments between S_1 and S_2,* i.e., $\mathrm{Align}(S_1, S_2)$*. Then, the* optimal anchored alignment *of S_1 with S_2, written* $\mathrm{Align}_{opt}(S_1, S_2)$*, is the one of $\{\theta \in \Theta \mid \forall \theta_1 \in \Theta, \mathrm{Sim}(\theta) \geq \mathrm{Sim}(\theta_1)\}$.*

$$\begin{tikzcd} & & C[1] \arrow[d] & \ldots \arrow[l] \arrow[d] & \mathbf{C[t]} \arrow[l] \arrow[r] \arrow[d, "anchor"] & \ldots \arrow[r] \arrow[d] & C[m] \arrow[d] & & \\ S[1] & \ldots \arrow[l] & S[j] \arrow[l] & \ldots \arrow[l] & S[i] \arrow[l] \arrow[r] & \ldots \arrow[r] & S[k] \arrow[r] & \ldots \arrow[r] & S[n] \end{tikzcd}$$

Figure 6: Anchored sequence matching schema

3.2 Anchored Sequence Matching

We define anchored sequence matching to select a *counterpart instruction* from the local function, that matches well against the trapping instruction, taking the contexts of these instructions into account. First, we select a set of *anchor points*, likely candidates for the counterpart instruction from the local function. Next, starting from each *anchor point*, we compute the optimal match between the preceding and following fragments of the code snippet and of the local function, respectively. Finally, we select the anchor point with the maximal overall similarity value as the counterpart instruction.

Algorithm 1 presents the anchored sequence matching algorithm. Lines 2-6 perform some initializations, including splitting the code snippet into three parts: the part before the trapping instruction, *C_left*, the trapping instruction, *trap*, and the part following the trapping instruction, *C_right*. *C_left* is reversed, because we want to match the sequence to the left of the trapping instruction backwards from the trapping instruction (*cf.* Figure 6). Lines 8-22 then iterate over the possible anchor points. For each one, the local function is split into *L_left*, *anchor*, and *L_right*, with *L_left* reversed (lines 9-12), analogous to *C_left*. Optimal anchored alignment is then used via the function **align** to match *C_left* against *L_left*, and *C_right* against *L_right*, and to obtain the resulting similarity score (lines 13-14). We also compute the similarity score of the trapping instruction and the chosen anchor point (line 15). If the sum of these scores is greater than the best score recorded so far, then the current anchor point is recorded as the best match (lines 16-21). The final result is then the best collected anchor point (line 23).

Algorithm 1 The anchored sequence matching algorithm

```
 1: function ASM(List $C$, List $L$, List $anchors$, Int $t$)
 2:     best_anchor ← −1
 3:     max_score ← $min\_value$
 4:     C_left ← reverse($C[1..t-1]$)
 5:     trap ← $C[t]$
 6:     C_right ← $C[t+1..C.length]$
 7:
 8:     for i = 1 → $anchors.length$ do
 9:         L_left ← reverse($L[1..t-1]$)
10:         anchor ← $L[i]$
11:         L_right ← $L[t+1..L.length]$
12:
13:         left_score ← align($C\_left$, $L\_left$)
14:         right_score ← align($C\_right$, $L\_right$)
15:         anchor_score ← score($trap$, $anchor$)
16:         total_score ← $left\_score + right\_score + anchor\_score$
17:
18:         if ($total\_score > max\_score$) then
19:             max_score ← $total\_score$
20:             best_anchor ← $i$
21:         end if
22:     end for
23:     return $best\_anchor$
24: end function
```

3.3 Anchored Alignment

Anchored sequence matching uses anchored alignment, $Align(\widehat{C}, \widehat{L})$, to match the code snippet substrings $\widehat{C}$ before and after the trapping instruction against the context $\widehat{L}$ of each proposed anchor point. Algorithm 2 shows a straightforward recursive implementation. For each pair of nonempty sequences this algorithm takes the maximum value produced by three possibilities: matching the heads of each sequence with each other and the tails of each sequence recursively with each other, or matching the head of one sequence with a gap and the tail of that sequence recursively with the other sequence. The remaining lines address the empty sequence cases. If the code snippet substring is empty (line 3), the result is 0, reflecting the fact that it need only be matched against a prefix of the local function substring. On the other hand, if the local function substring is empty (line 4), the result is the product of the length of the remaining code snippet substring and the gap score, effectively matching the remaining code snippet substring to gaps.

Algorithm 2 A recursive anchored alignment algorithm

```
 1: function align_rc(List $\widehat{C}$, List $\widehat{L}$)
 2:     match ($\widehat{C}$, $\widehat{L}$) with
 3:     | ([], _) → 0
 4:     | (_, []) → $|\widehat{C}| \times gap\_score$
 5:     | (c::ct, l::lt) →
 6:         let gap_score  = score(⊥, ⊥) in
 7:         let head_score = score(c, l)   in
 8:         let no_gap = align(ct, lt) + head_score in
 9:         let cs_gap = align(ct, $\widehat{L}$) + gap_score in
10:         let lk_gap = align($\widehat{C}$, lt) + gap_score in
11:             return max(no_gap, cs_gap, lk_gap)
12: end function
```

The recursive algorithm constructs a three-way tree of execution, in which each branch has height at most $|\widehat{C}| + |\widehat{L}|$. $|\widehat{C}|$ has fixed size, and except when the local function is very small, $|\widehat{C}| \leq |\widehat{L}|$. The complexity is thus $O(3^{|\widehat{L}|})$.

The recursive algorithm recomputes the scores of many possible alignments. To avoid this inefficiency, we use dynamic programming [3], as shown in Algorithm 3. This algorithm populates an array *dpm* of size $(|\widehat{C}| + 1) \times (|\widehat{L}| + 1)$. The value stored in $dpm[i][j]$ represents the similarity of the best alignment between $\widehat{C}[1..i]$ and $\widehat{L}[1..j]$. As anchored alignment requires matching the complete code snippet substring but only a prefix of the local function substring, we take the best value found anywhere in the row $dpm[|\widehat{C}|][*]$ as the result. This is in contrast with the dynamic programming implementation of global alignment [22], in which the iterative computation is the same, but the result is $dpm[|\widehat{C}|][|\widehat{L}|]$.

The dynamic programming algorithm starts by initializing the zero column of every row with a multiple of the gap score (lines 6-8), reflecting the need to match the elements of the code snippet substring against gaps when the local function substring runs out. It then initializes the zero row of every column with a multiple of the gap score (line 10-12), reflecting the cost of pushing the initial match of the code snippet substring past a prefix of the local function substring, which requires matching the elements of that prefix to gaps. The remainder of the algorithm iterates over the elements of the array, starting from the elements with the smallest indices, representing the early parts of each sequence

Algorithm 3 Anchored alignment by dynamic programming

```
 1: function Align_dp(List $\widehat{C}$, List $\widehat{L}$)
 2:     // init matrix with zero
 3:     dpm ← make_matrix(0..$|\widehat{C}|$, 0..$|\widehat{L}|$, 0)
 4:     gap_score ← score(⊥, ⊥)
 5:
 6:     for i = 1 → $|\widehat{C}|$ do
 7:         dpm[i][0] ← $i * gap\_score$
 8:     end for
 9:
10:     for j = 1 → $|\widehat{L}|$ do
11:         dpm[0][j] ← $j * gap\_score$
12:     end for
13:
14:     for i = 1 → $|\widehat{C}|$ do
15:         for j = 1 → $|\widehat{L}|$ do
16:             head_score ← score($\widehat{C}[i]$, $\widehat{L}[m]$)
17:             dpm[m][n] ← max (
18:                 $dpm[i-1][j-1] + head\_score$,
19:                 $dpm[i-1][j] + gap\_score$,
20:                 $dpm[i][j-1] + gap\_score$)
21:         end for
22:     end for
23:     return score_opt($dpm[|\widehat{C}|]$)
24: end function
```

and moving towards the elements with the largest indices, representing the later parts of the sequences. For each pair of offsets, the algorithm considers the possibilities that the sequence elements represented by the current position match (line 18), or that one or the other is matched against a gap (lines 19-20). In each case, the matching score or the gap score is added to the previously computed value found in the array at the position corresponding to the effect of the match on the two subsequences. The result for the current array element is the maximum result produced by any of these three cases. Once the array is filled, the result is the largest score anywhere in the row representing the end of the code snippet substring. The complexity of the algorithm is proportional to the size of the array *i.e.*, $O(|\widehat{C}| \times |\widehat{L}|)$.

We then further optimize the algorithm by reducing the number of columns in the matrix, representing the distance into the local function substring for which it is worth considering a match. Indeed, in the tail of $\widehat{L}$, the penalty of the leading gap alignment could eventually outweigh even a perfect alignment score for the remaining code snippet substring. Therefore, the optimal alignment can only involve instructions found within a certain distance R from the start of $\widehat{L}$. Let *max_score* be the maximal alignment value for two elements, *i.e.*, the score when the elements match perfectly, *gap_score* be the alignment value if either element is a gap, and *min_score* be the minimal alignment value, *i.e.*, the score when two elements are matched, but they are vastly different. Then, R is the smallest value such that $|\widehat{C}| \times min\_score > |\widehat{C}| \times max\_score + (R - |\widehat{C}|) \times gap\_score$, *i.e.*, $R > |\widehat{C}| \times ((min\_score - max\_score + gap\_score)/gap\_score)$, since *gap_score* is negative. This inequality says that a complete mismatch of each element of the code snippet substring at the beginning of the local function substring could provide a higher alignment score than any perfect alignment that appears later in the sequence, beyond a sequence of gaps. Thus, we can just populate the matrix up to column R, instead of the full length $|\widehat{L}|$. The inequality also shows that R is of the same order of magnitude as $|\widehat{C}|$, as the least R satisfying the inequality is bounded by a constant multiple of $|\widehat{C}|$.

This optimization reduces the complexity of sequence alignment to $O(|\widehat{C}| \times R)$. Since ($min\_score$ − max_score + gap_score) / gap_score is a positive constant, the complexity becomes $O(|\widehat{C}|^2)$. Because the Linux kernel limits the length of the code snippet substring to a small constant (64 bytes), $O(|\widehat{C}|^2)$ amounts to a constant.

The anchored sequence matching algorithm uses the dynamic programming based sequence alignment algorithm twice per anchor point. As the complexity of the latter is constant, due to the restricted size of the code snippet, the overall complexity of Anchored Sequence Matching is linear in the number of anchor points.

4. DESIGN DECISIONS

The matching algorithm is independent of how anchor points are selected and how the score associated with a match between two elements is determined. We now address these issues. We also consider the need to normalize the input in some cases and address the possibilities of ties.

4.1 Anchor Point Selection

The complexity of the Anchored Sequence Matching algorithm is proportional to the number of anchor points. The accuracy of the algorithm also depends heavily on the selection of anchor points. All suspicious points in the source sequence should be included as anchor points to ensure the soundness of the algorithm. However, we found that taking all points in the source sequence as possible anchor points hurts the accuracy, as some incorrect anchor points could obtain higher scores than the true counterpart point.

Our experiments showed that the counterpart instruction is often consistent with the trapping instruction in three ways: 1). The type of the opcode 2). The addressing modes of the operands 3). The offsets used in indirect addressing modes. For example, the instructions <mov 0x68(%rcx), %rcx> and <mov 0x68(%rax),%rax> have the same opcode type (mov), the same addressing mode (an indirect memory access for the source operand and a direct register access for the target operand), and the same offset of any indirect address (0x68), although they use different registers. Furthermore, we consider two instructions to be of the same type even if they operate on operands of different sizes (*e.g.* movl *vs.* movq) or their triggering conditions are opposite (*e.g.* je *vs.* jne).

We define a function $compare(i_1, i_2)$ that compares two instructions i_1 and i_2 based on the above criteria. The result is a triple $\langle type, mode, offset\rangle$ where each element corresponds to a binary value that represents the comparison result with respect to he corresponding criterion, *e.g.*,

$$compare(\frac{\text{<mov 0x68(\%rcx),\%rcx>}}{\text{<mov 0x68(\%rax),\%rax>}}) = \langle 1,1,1\rangle$$

There are usually multiple anchor point candidates in the source sequence. To prioritize their selection, we build three queues based on the above criteria. The high-priority queue consists of the anchor points that satisfy all three criteria, *i.e.*, $\{i \mid compare(i, t) = \langle 1, 1, 1\rangle\}$. The medium-priority queue consists of the ones that have the same instruction type as the trapping instruction, *i.e.*, $\{i \mid compare(i, t) = \langle 1, *, *\rangle\}$. And the low-priority queue consists of all the instructions in the local function. For the matching, we select the first non-empty queue following the order of the priority.

4.2 Scoring Function Design

The scoring function should measure the similarity of two related elements as well as the unlikelihood for the matching of two unrelated elements. The comparison between two elements can result in a *match*, a *mismatch*, or a *gap match* (*i.e.*, either element is empty). A *match* should have a higher score than a *mismatch*, and we previously argued that a gap match should have a negative score, to eliminate the match $(\perp, \perp)$ in any optimal solution [22]. We propose two possible scoring functions, differing in how much information about the instructions that they take into account.

Our first scoring function based on opcode, *score*, considers two instructions to be a *match* if they have the same type of opcode, *i.e.*, $compare(i_1, i_2) = \langle 1, *, *\rangle$, and any other comparison between two instructions to be a *mismatch*. We assign the scores 2 for a *match*, -1 for a *mismatch*, and -1 for a *gap match* [22]. In practice, we have observed that the *gap_match* value should not be lower than the *mismatch* value, or the algorithm will almost never assign a gap.

Our second scoring function based on both opcode and operands, *score′*, further refines the *match* case, following the comparison criteria proposed for the anchor point selection. For those instructions with both source and target operands, we give one point bonus if the addressing modes of the operands of the two instructions match, and another point if the offsets of indirect addressing modes match as well:

$$score'(i_1, i_2) = \begin{cases} 2 + mode + offset, & \text{if } type = 1 \\ score, & \text{otherwise} \end{cases}$$
$$\text{where } \langle type, mode, offset\rangle = compare(i_1, i_2)$$

4.3 Breaking Ties

Our algorithm selects anchor points and runs the sequence matching algorithm for each of them in order to identify the anchor point with the highest matching score. In case of a tie, we increase the score of each result with the best score by $score'(trap, anchor)$, thus doubling the weight of the anchor point. The strategy favours the anchor point that is more similar to the trapping instruction. If this computation again produces a tie, we then select the anchor point that occurs earlier in the local function. This strategy is based on the observation that *e.g.*, if the oops is due to a dereference of a pointer that is NULL already at the beginning of the function, then only the first dereference attempt will be executed.

4.4 Input Normalization

The disassembled code snippet or local function may contain instructions such as <nop> (no operation), or <xchg %ax,%ax> (exchange the values of registers), that have no impact on the result of the execution, but that may *e.g.*, improve memory alignment. We refer to these instructions as *junk* instructions. The junk instructions might interfere with our matching, since they could unnecessarily incur the cost of a mismatch or a gap match if the compiler of the victim kernel and the compiler of the local function do not introduce them according to the same strategy. We thus remove all junk instructions before matching.

5. EVALUATION

We have implemented our approach as a tool named *Oopsa*.[3] We now evaluate *Oopsa*, presenting our experi-

[3] *Oopsa* rhymes with a French phrase "où ça?" (where is it?).

Table 2: Distribution of Trapping Instructions

opcode	mov*	cmp*	test	add	and	ud2	inc	bts
count	73	14	5	2	2	2	1	1

mental data and settings, the performance benchmarking, and an analysis of the failure cases.

5.1 Experimental Data

We have selected 100 kernel oops samples, of which 90 come from the kernel oops repository [2] and 10 are from the attachment of bug reports in the Linux kernel bugzilla [11]. *Oopsa* requires that a kernel oops have the following properties: I). It should contain a code snippet that can be properly decoded. II). Gcc should be able to generate debugging information mapping between the source code and assembly code of the crashing function. Among the kernel oopses satisfying these properties, we randomly selected 100 samples. The distribution of trapping instructions among these 100 kerne oopses is shown in Table 2, with 73 being variants of the *mov* instruction. Furthermore, in 95 out of 100 cases, the trapping instruction involves a memory-based operation. In 19 cases an operand is a constant, which can make the instruction more distinguishable. Finally, the number of instructions for the code snippet is between 10 and 27, with 18 on average.

To evaluate *Oopsa*, we establish the ground truth for each kernel oops by manually inferring the offending line through the semantic matching (*cf.*, Section 2.2). This analysis requires up to several hours per oops, limiting the number of cases that we can consider. For some of the bugzilla samples, the offending line is given in the subsequent discussion.

5.2 Experimental Settings

Our experiments use a 64 bit machine(Debian 3.2.54-2), the easily accessible mainline source code from kernel.org corresponding to the version mentioned in the kernel oops, gcc version 4.7.2 (Debian 4.7.2-5), and a kernel configuration file generated by the standard command `make ARCH=i386|x86_64 defconfig`, specifying a 32 or 64-bit architecture, as indicated by the oops. Thus, our approach does not burden the developer with hunting down distribution-specific debugging kernels or recreating the exact build environment used to create the victim kernel. The default compiling configuration as stated previous allows us to generate the assembly code for almost all crashing functions in our samples. For those exceptional cases (7 out of 100), we then used the .config file of Debian 3.2.54-2 machine, compiled the source file directly and chose the default value for every prompt option, in order to generate the assembly code for the crashing function.

Our experiments measure both the accuracy and the running time of our solution. Accuracy is measured as the percentage of cases where the result is consistent with the established ground truth. Running time is measured as the average execution time per sample, over the 100 samples, including the time for parsing the input, the time for running the matching algorithm, and the time for generating the output. We measure the running time on a Mac mini with an Intel Core i5 2.5 GHz.

Our tool, *Oopsa*, is implemented in OCaml, consisting of the parsing of the kernel oops, the parsing of assembly code, and the matching algorithms. The implementation amounts to 1645 lines of OCaml code. This OCaml code is compiled into optimized native code for the evaluation.

Table 3: Performance with core settings

Scoring Function	Anchor Point Selection	
	No	Yes
opcode based	64% / 450ms	86% / 20ms
opcode-operand based	72% / 838ms	90% / 33ms

Table 4: Performance with peripheral settings

Input Normalization	Score Adjustment	
	No	Yes
No	90% / 33ms	91% / 33ms
Yes	91% / 31ms	92% / 31ms

5.3 Performance Benchmarking

Overall, *Oopsa* automatically identifies the correct offending line in 92 cases (92% accuracy), while the approach merely relying on the instruction pointer only identifies the correct offending line in 26 cases (26% accuracy). For these 26 cases, *Oopsa* works as well.

In Section 4, we proposed several designs concerning the anchor point selection, the scoring function, the matching score adjustment and the input normalization. We now evaluate the impact of these designs.

The algorithms for anchor point selection and for the scoring function affect the treatment of every oops, and thus we consider them together. Table 3 shows the accuracy and the running time for each possible combination. Using every instruction as an anchor point, combined with the scoring function that takes only the type of the opcode into account gives the worst accuracy (64%). Extending the scoring function to also take properties of the operands into account improves the accuracy (72%), but at a heavy performance penalty (>80%) since it requires more sophisticated parsing for each instruction. On the other hand, just with the operand type based scoring function, simply applying anchor point selection gives a greater improvement in accuracy (86%), because it eliminates anchor points that mislead the algorithm, and also greatly improves the running time (22 times faster), because the complexity of the matching algorithm is linear in the number of anchor points. Further augmenting with the type-operand based scoring function, we obtain the best accuracy (90%). We furthermore note that the observed improvements in the number of correct results are monotonic; the number of correct results improves, while no previously correct results become incorrect.

Table 4 shows the impact of augmenting the use of the anchor point selection algorithm and the scoring function based on both the opcode and properties of the operands with the elimination of junk instructions and the treatment of ties. These optimizations, although not essential, help *Oopsa* deal with certain corner cases and have little impact on the running time.

For all the results shown in Tables 3 and 4, we have applied the optimization that reduces the size of matrix considered in the underlying dynamic programming algorithm, as presented in Section 3.3. The optimization greatly improves the running time of our solution, regardless of the other settings. Without this optimization, the running times in *e.g.*, the bottom row of Table 3 rise to 4253ms and 112ms respectively, an increase of up to over 5 times.

5.4 Failure Case Analysis

In six cases, either the local function is organized is a way that is profoundly different from the code snippet, so that the sequence matching algorithm is misled, or there are several very similar subsequences in the local function, and the sequence matching cannot distinguish among them. These cases break our assumption that the neighborhood of the counterpart instruction is more similar to that of the trapping instruction than that of any other instruction. For example, in one case, the crash occurs one line before a branching statement. In the code snippet, the branch is implemented using the jne instruction, while in the local function, the branch is implemented using its opposite, je, changing drastically the instruction order. In another case, the oops is generated by a use of the BUG_ON macro, which occurs twice in the crashing function, in almost identical contexts. Both occurrences thus get the same score.

Second, our anchor point selection algorithm assumes that the opcode of the counterpart instruction is at least of the same type as the trapping instruction. In two cases, this assumption does not hold, leading to failure. For instance, in one case, the trapping instruction is a memory-access operation <movzwl 0x54(%rbx),%eax> followed by a test <test $0x2,%al>, while the counterpart instruction is a test <testb $0x2,0x54(%rbx)> combining both operations. As a result, the counterpart instruction is overlooked by the anchor point selection mechanism, since there are several candidates that match the trapping instruction better.

To address these cases, we plan to integrate more semantic analysis, such as data flow analysis, into the matching algorithm. One observation from the above example is that the <test $0x2,%al> instruction has a data dependency on the trapping instruction movzwl over the register %al. It should be noted that the test instruction is more identifiable, especially with a constant operand, since it appears less frequently than the variants of mov instruction. Thus, it may be more effective to find a counterpart instruction for the test instruction, instead for the trapping instruction, and to work back from there. We leave this as future work.

6. THREATS TO VALIDITY

In this section, we discuss the threats to the various types of validity of our solution.

Construct Validity. *Does the accuracy metric actually measure the accuracy of our solution?* The accuracy of the measurement depends on the ground truth. For each kernel oops sample, we establish the ground truth (its offending line) through both the assembly code matching and the semantic matching approaches. We found at least 3 pieces of evidence to support our assessment in each case. In addition, in certain samples from the Linux kernel bugzilla, the ground truth is given in the comments following the kernel oops, which reconfirms our assessment. There might, nevertheless, be some mistakes in our assessment.

Internal Validity. *Are the underlying assumptions of our sequence matching algorithm valid in practice?* Our main assumptions are: I). If we find the counterpart instruction for the trapping instruction, then we can find the offending line. II). The counterpart instruction is situated in a neighborhood that is the most similar to the one of the trapping instruction. The first assumption depends on the functionality of gdb, which holds for all cases in our experiment. The second may not hold if the instruction sequence of the crashing function that led to the oops is significantly different from that of the mainline kernel. This can occur if the configuration options or source code are different; Linux is open source software, and anyone can freely recompile the kernel under different configuration options and even modify the source code. Alternatively, as described in Section 5.4, there is a risk that the algorithm might be misled by multiple very similar regions of code within the crashing function.

External Validity. *To which extent does the solution generalize to new kernel oopses, or to different platforms?* We argue that the samples in our experiments are representative, since they are extracted from well recognized sources: the kernel oops repository and the Linux kernel bugzilla. We are not aware of any other resources that provide more representative kernel oopses.

There are currently around 4400 kernel oopses with code snippets in the kernel oops repository. We obtained 92% accuracy on the 90 randomly selected samples from the repository. According to the sample size calculator,[4] our solution should work for at least 95%(−5.55) of all qualified kernel oopses. Therefore, we have a relatively high confidence that our solution works for any kernel oops that satisfies the properties listed in Section 5.

Furthermore, our sequence matching algorithm is primarily designed to locate a counterpart point in one sequence for a point of interest in another sequence. It is independent of the platform and the programming language. Therefore, it should be applicable to any scenario having a similar goal.

7. RELATED WORK

Error Report Debugging. WER [15] (Windows Error Reporting) is a distributed, postmortem debugging system that collects error reports from the Windows operating system as well as over 7000 third-party applications. Due to the large deployment of Windows, one of the primary design goals of WER [6] is to prioritise the error reports. For this, WER groups the error reports into buckets based on a series of criteria, such as the crashing program name, the exception code, and the trapping instruction offset. ReBucket [5] clusters WER crash reports based on call stack matching.

The Linux kernel oops repository [2] receives several hundred reports per day. We have previously done a statistical analysis of kernel oopses [8]. In this paper, we focus on helping developers debug a particular kernel oops, individually.

Several researchers have used fault injection to study the resistance of the kernel to crash conditions generating oopses [7, 25]. Our work is different, in that it aims to help developers debug kernel oopses. Our sample data also come from the real world, instead being generated by fault injection.

Software Fault Localization. Fault localization is the activity of identifying the exact locations of program faults [24]. Our approach identifies the offending line, which is the direct cause of an error report, but is not necessarily the root cause of a bug. On the other hand, our approach does not require the execution of the crashing program, or the presence of test cases, unlike most of the fault localization techniques.

[4]http://www.surveysystem.com/sscalc.htm

Sequence Alignment. In biology, global sequence alignment was first proposed by Needleman and Wunsch [16] in 1970, in order to search for similarities in the amino acid sequence of two proteins. Later, Smith and Waterman [21] introduced the problem of local alignment. Both alignments are implemented based on the systematic study of dynamic programming by Belleman [3] in 1957. Finally, a number of authors have studied the question of how to construct a good scoring function for sequence comparison, to meet the matching criteria of different scenarios [1, 10].

Our problem is different from both the global and local sequence alignment. Given a code snippet C and a local function L, we seek to find a substring $\widehat{L}$ of L that has the best global sequence alignment with C. In addition, we give priority to the matching of the *counterpart* instruction — a point of interest within the source sequence that is likely to match the best with the trapping instruction in the snippet.

Sequence alignment has already been applied in software engineering. Han *et al.* [9] applied sequence alignment to call traces, while Lo *et al.* [14] did the same for program execution traces. Levenshtein distance [13] uses global sequence alignment in measuring the minimal editing effort required to change one text to another one. Several efforts have been made to track the evolution of other software artifacts, such as program execution traces [18], or variables [19], using the global sequence alignment.

8. CONCLUSION

In this paper, we have shown how to use approximate sequence matching to identify the offending line of a kernel oops. Our approach is fully automatic, from extracting the code snippet from the oops and obtaining the source code to returning the offending line. As such, our approach has great potential to help developers take advantage of the recently established kernel oops repository.

In future work, we will consider how to improve the accuracy of our approach, possibly by applying analyses such as dataflow analysis to the code snippet. We will also evaluate our approach on a larger set of oopses, and consider whether our approach could be relevant to other kinds of software.

9. ACKNOWLEGEMENT

We thank David Lo for his advice and feedback.

10. REFERENCES

[1] S. F. Altschul. A protein alignment scoring system sensitive at all evolutionary distances. *Journal of Molecular Evolution*, 36:290–300, 1993.

[2] A. Arapov. Kernel oops repository, Sept. 2012.

[3] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] Bug report in bugzilla: http://bugzilla.kernel.org/show_bug.cgi?id=27492.

[5] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *ICSE*, pages 1084–1093, 2012.

[6] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, pages 103–116, 2009.

[7] W. Gu, Z. Kalbarczyk, K. Ravishankar, and Z. Yang. Characterization of Linux kernel behavior under errors. In *DSN*, pages 459–468, June 2003.

[8] L. Guo, P. Senna Tschudin, K. Kono, G. Muller, and J. Lawall. Oops! what about a million kernel oopses? Technical Report RT-0436, Inria, June 2013.

[9] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

[10] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. In *Proceedings of the National Academy of Science*, volume 87, pages 2264–2268, USA, Mar. 1990.

[11] Kernel bug tracker: https://bugzilla.kernel.org/.

[12] M. Kerrisk. Kernel submit: Improving tracing and debugging, 2012. https ://lwn.net/Articles/514898/.

[13] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, pages 707–710, 1966.

[14] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *ACM SIGSOFT FSE*, pages 265–275, 2006.

[15] Microsoft MSDN. Windows error reporting.

[16] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[17] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: ten years later. In *ASPLOS*, pages 305–318, Mar. 2011.

[18] M. K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *ASE*, pages 241–252, 2006.

[19] M. K. Ramanathan, S. Jagannathan, and A. Grama. Trace-based memory aliasing across program versions. In *FASE*, pages 381–395, 2006.

[20] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *EuroSys*, 2011.

[21] F. S. Temple and S. W. Michael. Identification of common molecular sequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[22] M. Tompa. Lecture notes on biological sequence analysis. Technical Report 2000-06-01, Department of Computer Science and Engineering University of Washington, Winter 2000.

[23] L. Torvalds. http://www.kernel.org/doc/ Documentation/oops-tracing.txt.

[24] W. E. Wong and V. Debroy. A survey on software fault localization. Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, Nov. 2009.

[25] T. Yoshimura, H. Yamada, and K. Kono. Is Linux kernel oops useful or not? In *HotDep*, Oct. 2012.