Patch Specifications via Product Programs

Cristian Cadar
Department of Computing
Imperial College London
London, UK
c.cadar@imperial.ac.uk

Daniel Schemmel
Department of Computing
Imperial College London
London, UK
d.schemmel@imperial.ac.uk

Arindam Sharma
Department of Computing
Imperial College London
London, UK
arindam.sharma@imperial.ac.uk

***Abstract*—Code patches are the basic blocks of software evolution and several testing and analysis techniques have been proposed to validate them. However, due to lack of specifications, most of these techniques focus on generic errors, such as crashes.**

In this vision paper, we propose to adopt product programs as a practical means of writing patch specifications that could be checked using existing testing techniques.

Future work will investigate the feasibility of automatically generating product programs for real-world code patches, the ease of writing useful patch specifications, and the integration of such patch specifications with existing testing techniques.

***Index Terms*—code patch, specification, product program**

I. Introduction

We cannot check what we cannot specify. Despite the significant progress we have seen in software testing and analysis, with powerful techniques such as symbolic execution [3], fuzzing [19] and search-based testing [17] becoming more and more scalable, these techniques have primarily been applied to find generic bugs, such as crashes. The primary reason for this limitation is that most widely-used software systems lack specifications.

Recent progress on specifying the correctness of software has been significant. Researchers have specified and verified several key pieces of infrastructure, such as compilers [13] and OS kernels [10]. However, the effort of writing specifications for these systems is tremendous, often involving years of PhD-level expertise and specifications that are several times larger than the code itself.

Furthermore, these specifications need to be kept in sync with the evolution of the code, with the required changes not always modular. This is perhaps one of the most important reasons for which these systems have not become more mainstream and have not attracted a broader developer base. To evolve these systems, developers would not only need a good understanding of the code, but also of the (often much larger) specification, and would need to be proficient in updating both.

In this vision paper, instead of targeting whole-system specifications, we investigate a more lightweight direction focused on (partial) patch specifications. Code patches are the basic blocks of software evolution and many critical bugs and security vulnerabilities—Heartbleed [8] being just a famous recent example—are introduced via patches, rather than being part of the core system since its early stages.

Given an appropriate mechanism for writing software patches, would developers take advantage of it? Would patch specifications, rather than whole-system ones, be easier for developers to write? Could they benefit real-world systems for which no specifications have been made available? Could they be used by existing software testing techniques such as symbolic execution and fuzzing?

We believe that several requirements would make it more likely to have a positive answer to the questions above. First, patch specifications should be allowed to be independent of any previous specifications available in the system. That is, developers should not be required to understand previous specifications (but they could choose to do so to reuse them), allowing them to keep their focus on the context of the patch.

Second, developers should be able to decide to write anything between full patch specifications (describing all the new behaviours introduced by the patch) to partial patch specifications (describing some of the key new behaviours) to no specifications at all. Our hope is to have patch specifications adopted incrementally, as part of real-world software.

Third, no manual work should be required to support such patch specifications. The system should provide an automated way to refer to the state of the pre- and post-patch versions.

Fourth, these specifications should be *executable*, in the sense that they should be expressed via code, in the same programming language as the one in which the software is written. This would avoid requiring developers to learn an unfamiliar specification language.

Finally, these patch specifications should benefit existing testing techniques, such as symbolic execution and fuzzing, in an "out-of-the-box" manner. That is, existing tools, say AFL++ [7] and KLEE [3], should be able to use these specifications without any modifications.

How would developers write patch specifications? At a high-level, developers would be given simultaneous access to the variables (or memory locations more generally) in both the pre-patch and post-patch versions of the program. As most variables would be shared between the two versions, they would be disambiguated by e.g., adding the suffix `_prev` to the pre-patch version. With access to both sets of variables, developers could write arbitrary code that states properties between them.

We have identified *product programs* [1] as a promising mechanism for writing patch specifications. Product programs essentially merge several program versions into a single program, enabling the use of standard analysis techniques, such

```
1 int Fn;
2 if (n <= 1)
3   Fn = 1;
4 else {
5   int Fn_2 = 1; // F(n-2)
6   int Fn_1 = 1; // F(n-1)
7   Fn = Fn_1 + Fn_2;
8
9   for (int i = 2; i < n; i++) {
10    Fn_2 = Fn_1;
11    Fn_1 = Fn;
12    Fn = Fn_1 + Fn_2;
13  }
14 }
```

Fig. 1. Code aiming to compute the n-th Fibonacci number.

```
   if (n <= 1)
-    Fn = 1;
+    Fn = n;
   else {
-    int Fn_2 = 1; // F(n-2)
+    int Fn_2 = 0; // F(n-2)
```

Fig. 2. Patch for the code in Fig. 1.

as symbolic execution and fuzzing, to reason about properties that connect the different program versions.

Creating executable product programs for real programs and code patches is challenging, but if successful, they could enable existing techniques to check the patch specifications written by developers.

II. Example

To illustrate our idea for patch specifications, consider the code in Fig. 1, which aims to compute the n-th Fibonacci number. Fibonacci numbers are typically defined by $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$, so the generated sequence should be $F(0) = 0, F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, ...$ Instead, the code in Fig. 1 generates the sequence $F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5, F(5) = 8, ...$

Suppose the developer notices the incorrect base case and applies the patch in Fig. 2. Knowing that the patch aims to shift the sequence to the right by one, they could write the patch specification:

$$\text{Fn} = \text{Fn\_1}_{prev}$$

This would be placed as an `assert` statement just before the end of the `else` case. (Note that while this is a strong patch specification, it is nevertheless a partial specification, as it ignores the change in behaviour for $n = 0$.)

III. Product Programs

Developers will only write patch specifications if they can see an immediate benefit. Therefore, we would like existing testing techniques to be able to use these patch specifications "out of the box", searching for any inputs that may violate the specifications.

```
1 int Fn_prev; // previous version
2 int Fn;   // current version
3
4 if (n <= 1) {
5   Fn_prev = 1;
6   Fn = n;
7 }
8 else {
9   int Fn_2_prev = 1; int Fn_2 = 0;
10  int Fn_1_prev = 1; int Fn_1 = 1;
11
12  Fn_prev = Fn_1_prev + Fn_2_prev;
13  Fn = Fn_1 + Fn_2;
14
15  for (int i = 2; i < n; i++) {
16    Fn_2_prev = Fn_1_prev; Fn_2 = Fn_1;
17    Fn_1_prev = Fn_prev; Fn_1 = Fn;
18
19    Fn_prev = Fn_1_prev + Fn_2_prev;
20    Fn = Fn_1 + Fn_2;
21  }
22
23  assert(Fn == Fn_1_prev); // patch spec
24 }
```

Fig. 3. Product program for the code in Fig. 1 and the patch in Fig. 2.

To accomplish this, we propose to adopt the idea of *product programs* [1], [4], to create a program that merges the pre- and post-patch versions, allowing the patch specification to refer to variables in both versions.

At a high-level, product programs give unique names to the variables in the two versions and then interleave the instructions of the two versions. A product program for our example is shown in Fig. 3. Each version has its own variables: the pre-patch version has all its variables renamed to add the suffix `_prev`. However, for readability, we show a product program where the read-only argument `n` and the iteration counter `i` are not duplicated, and which exploits the fact that the loop has the same number of iterations in both versions.

With this product program in place, one can write the specification discussed above, in an executable form, by adding the assert shown on line 23:

```
assert(Fn == Fn_1_prev);
```

Of course, these patch specifications would be written directly in the post-patch program (in our example after line 13 in Fig. 1), with the product program constructed automatically later on, during the testing stage.

IV. Preliminary Experience

We have experimented with patch specifications backed-up by product programs on several patches from the COREBENCH [2] suite, a collection of complex real-world patches from popular open-source programs. In particular, we have investigated writing patch specifications for three patches from GNU COREUTILS, a suite of core utility programs such as `ls` and `mkdir`, installed on most UNIX-based systems. We choose to present three of these patches, each illustrating a different scenario in which patch specifications could prove

```
1  - if (*linkname == '/')
2  + if (IS_ABSOLUTE_FILE_NAME (linkname))
3      return xstrdup (linkname);
4
5  - char const *linkbuf = strrchr (name, '/');
6  - if (linkbuf == NULL)
7  + size_t prefix_len = dir_len (name);
8  + if (prefix_len == 0)
9      return xstrdup (linkname);
10
11 - size_t bufsiz = linkbuf - name + 1;
12 - char *p = xmalloc (bufsiz + strlen (linkname) +
       1);
13 - strncpy (p, name, bufsiz);
14 - strcpy (p + bufsiz, linkname);
15 + char *p = xmalloc (prefix_len + 1 + strlen (
       linkname) + 1);
16 + stpcpy (stpncpy (p, name, prefix_len + 1),
       linkname);
17   return p;
18 }
```

Fig. 4. Patch in `ls` (COREBENCH #14a), ignoring a declaration moved down.

helpful: refactoring; introduction of an error case and partial refactoring; and output changes.

Refactoring in `ls`. Fig. 4 shows a patch in the `ls` tool which is intended to be a refactoring—the commit message says *"Do not hard-code '/'. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/stpncpy in place of strncpy/strcpy."*

Refactoring patches are an excellent example of a scenario for which patch specifications are easy to write and effective in flagging any issues. For this patch, the developer could simply state just before the return statement on line 17 that the computed string `p` does not change across versions:

```
assert(0 == strcmp(p, p_prev));
```

In addition, one could also ensure that the early returns are triggered in the same way across versions. For instance, the following assert could be added just before line 2 (and a similar one before line 8):

```
assert((IS_ABSOLUTE_FILE_NAME (linkname)) ==
    (*linkname_prev == '/'));
```

This refactoring in `ls` is incorrect. In particular, the post-patch version produces a different output in some cases, e.g. when `name` and `linkname` are `/a` and `x`. We were able to construct a small driver that invokes the function containing the patch, and then ran AFL++ and KLEE on it. Both tools managed to generate inputs that violated our specification.

The bug made it into an official release of COREUTILS, after which it was reported by a user [15] and fixed in time for the next release [14]. Our hope is that with a simple patch specification like the one above, such errors could be more easily detected at development time.

As a further check, we incorporated the fix and ran AFL++ and KLEE again. We were surprised to see that the tools still flagged divergences. For instance, setting `name` and `linkname` to `/x//y` and `a` produces the output `/x//a` in the original version and `/x/a` in the fixed version. While our

```
1    case string:
2  - i = 0;
3  - cp = v->u.s;
4  - neg = (*cp == '-');
5  - if (neg)
6  + {
7  +  intmax_t value = 0;
8  +  char *cp = v->u.s;
9  +  int sign = (*cp == '-' ? -1 : 1);
10 +  if (sign < 0)
11       cp++;
12    do {
13      if (ISDIGIT (*cp)) {
14 -    i = i * 10 + *cp - '0';
15 +    {
16 +      intmax_t new_v = 10 * value + sign * (*cp
       - '0');
17 +      if (0 < sign
18 +      ? (INTMAX_MAX / 10 < value || new_v < 0)
19 +      : (value < INTMAX_MIN / 10 || 0 < new_v))
20 +       error (EXPR_FAILURE, 0,
21 +            (0 < sign
22 +            ? _("integer␣is␣too␣large:␣%s")
23 +            : _("integer␣is␣too␣small:␣%s")),
24 +            quotearg_colon (v->u.s));
25 +      value = new_v;
26 +    }
27      else
28            return false;
29    }
30    while (*++cp);
31    free (v->u.s);
32 - v->u.i = i * (neg ? -1 : 1);
33 + v->u.i = value * sign;
34    v->type = integer;
35    return true;
```

Fig. 5. Fragment of the patch in `expr` (COREBENCH #22a). Original formatting changed for better readability.

understanding is that this change does not introduce any issues in `ls`, it is nevertheless important to be aware of and correctly document the change. If it is intentional, the specification could compare the two paths with a custom function that allows for different representations of the same path.

New error case and partial refactoring in `expr`. The second patch we present is in the `expr` tool, in the code which converts a string to an integer. This patch added code to detect the case where the integer is too small or too large, in which case it exited with an error. To accomplish this, it also refactored this code. Part of the patch is shown in Fig. 5, with the new error case on lines 20–24.

Adding a specification for the case where the integer is out of range would have involved replicating the logic of the patch, which we found unhelpful. Instead, we added a simple specification at the end of the refactored function stating that the refactored code produces the same integer output when the error case is not triggered in the post-patch version:

```
assert(v->u.i == v_prev->u.i);
```

The refactored code introduced an error involving negative numbers. After constructing a small test driver, both AFL++ and KLEE generated inputs that violated our patch specification. The fix proposed by the developers after a user reported the

```
1 -     puts (p);
2       char *z = buf;
3 +
4 +     /* Write first number to buffer. */
5 +     z = mempcpy (z, p, p_len);
6 +
7 +     /* Append separator then number. */
8       while (cmp (p, p_len, q, q_len) < 0)
9         {
10 +       *z++ = *separator;
11          incr (&p, &p_len);
12          z = mempcpy (z, p, p_len);
13 -       *z++ = *separator;
14 -       if (buf_end - n - 1 < z)
15 +       if (buf_end - (n + 1) < z)
16            {
17              fwrite (buf, z - buf, 1, stdout);
18              z = buf;
19            }
20        }
21
22 -     if (buf < z)
23 -       fwrite (buf, z - buf, 1, stdout);
24 +     *z++ = *terminator;
25 +     fwrite (buf, z - buf, 1, stdout);
```

Fig. 6. Patch in `seq` (COREBENCH #8b), with some comments omitted.

introduced bug [6] was a one line change [5]. We incorporated this fix and used again AFL++ and KLEE: this time no inputs could be produced that violated the patch specification.

Output changes in `seq`. The last patch we are presenting, in `seq`, performs a bug fix that changes the output of the tool. For instance, the command `seq -s, 6 9` (instructing the tool to print the sequence from 6 to 9, with comma as a separator) would print in the pre-patch version:

```
6
7, 8, 9,
```

while in the post-patch version it would print: `6, 7, 8, 9`

The patch is shown in Fig. 6. The pre-patch version prints the first number followed by a new line, after which it has a loop which prints each number followed by a separator into an output buffer. The post-patch version starts by printing the first number into the output buffer, and then changes the loop iterations to print into the buffer first the separator and then the next number in the sequence. Our patch specification essentially encodes the fact that just after a separator is printed in the post-patch version, the contents of the output buffer of one version is the suffix of the contents of the output buffer of the other version. We achieved this by adding after line 10 the following assert, where `MIN` is a macro computing the minimum of its two arguments:

```
size_t min = MIN(z_prev - buf_prev, z - buf);
assert(0 == memcmp(z_prev - min, z - min, min));
```

and the following asserts after the while loop, which encode the difference introduced by the patch without relating the states of the two versions:

```
assert(buf_prev == z_prev
  || z_prev[-1] == *separator_prev);
assert(buf == z || z[-1] >= '0' && z[-1] <= '9');
```

As for the other patches, we constructed a test harness and then ran both AFL++ and KLEE: neither tool found any specification violations.

Our patch specifications and the test harnesses we constructed are both available as part of the paper artifact at https://srg.doc.ic.ac.uk/projects/patch-specs/ and https://doi.org/10.5281/zenodo.7591940.

V. Discussion and Future Work

Our preliminary evaluation has shown that in several scenarios, patch specifications can be written in a lightweight manner, and find inadvertently introduced bugs. The key difference between a regular functional specification and a patch specification is that the latter encodes what has changed in the program. By definition, this is exactly what the developer has been working on, so they are in the best possible context to write a patch specification.

Patches typically change a subset of program behaviour; for refactorings, this subset is empty. Therefore, perhaps the simplest way to approach patch specifications is to focus on the case where the behaviour should be unchanged, as we did for the `ls` and `expr` patches. This typically means that the output (to the program, function, or code fragment) should stay the same in those cases. Such patch specifications are quite easy to write, but can be effective in finding common types of bugs, where the patch mistakenly changes program behaviour in too many or too few cases. Importantly, note that these specifications can be written at a fine level of granularity, involving intermediate program states, rather than only in terms of the final output, as it is usually the case in differential testing [16].

Patch specifications encoding output changes, such as the one for `seq`, are more difficult to write, but can nevertheless be easier to state than regular specifications. Of course, it is easy to think of cases where regular specifications are easier to write than patch specifications, in which case the former should be preferred. Patch specifications are not meant to replace, but rather complement regular specifications.

While in principle one could write a patch specification for any change, the most useful specifications are those that encode relationships between the two versions in a different way than the code itself (see the `expr` patch where we found it unhelpful to write a specification for the error case). In this paper, we have also assumed that specifications are written for individual patches. Of course, sometimes it may make more sense to write specifications for a group of patches (e.g. those forming a pull request) or involving non-consecutive versions (as we did for the `ls` patch after incorporating the fix).

Patch specification can be useful in understanding a patch, but their main utility is in conjunction with an automatic testing or verification tool. To this end, they depend on the ability to construct product programs. To date, product programs have not been shown to scale to large codebases. Whether they can be generated automatically for real-world patches of large programs is still an open question. Challenges include keeping the execution of the two versions synchronised (e.g.

not repeating common function calls in each version, handling loops with different numbers of iterations across versions, etc.), determining the program points associated with the patch specification in the two program versions (if the patch is written in the current version, it may be difficult to find the right program point in the previous version, and developer assistance might be needed for more complex patches), and dealing with environmental side effects, such as those related to input and output streams (so that the two versions do not incorrectly interfere with one another), among others. Nevertheless, recent work on modular product programs [4] and our current experience building a prototype that constructs product programs for C code are promising.

VI. Related Work

While we are unaware of prior work directly targeting developer-written patch specifications, there is a large body of work on code-level specifications, e.g. [18], including work on integrating specifications and testing [9].

Product programs have been used in differential assertion checking [11] to determine whether errors are introduced by a new program version. While the framework could in principle handle the kind of patch specifications we envision, it focuses exclusively on generic errors such as buffer overflows. Reasoning is done using static analysis, which imposes different requirements on the construction of product programs.

In addition to product programs [1], [4], other forms of multi-version programs have been proposed in the past, e.g. in work on shadow symbolic execution [20] and multiversion interprocedural control flow graphs [12].

VII. Conclusion

Software patches are the basic blocks of software evolution and they should be comprehensively tested. In this paper, we have proposed the use of product programs as a practical means of writing patch specifications that could be checked using standard testing techniques such as fuzzing and symbolic execution. We have reported our promising initial experience writing specifications for complex patches and have identified several important directions for future work.

Acknowledgements

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement no. 819141).

References

[1] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Proc. of the 17th International Symposium on Formal Methods (FM'11)*, Jun. 2011.

[2] M. Böhme and A. Roychoudhury, "CoREBench: Studying complexity of regression errors"," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'14)*, Jul. 2014.

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[4] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 1, 2020.

[5] "expr bug fix." [Online]. Available: http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=6fc0ccf717c09d2dc941ee9d6622e7e987653eeb

[6] "expr bug report." [Online]. Available: https://lists.gnu.org/archive/html/bug-coreutils/2005-05/msg00189.html

[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. of the 14th USENIX Workshop on Offensive Technologies (WOOT'20)*, Aug. 2020.

[8] "Heartbleed bug," http://heartbleed.com/, Apr. 2014.

[9] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, Feb. 2009.

[10] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, Oct. 2009.

[11] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[12] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, May 2014.

[13] X. Leroy, "Formal verification of a realistic compiler," *Communications of the Association for Computing Machinery (CACM)*, vol. 52, no. 7, pp. 107–115, 2009.

[14] "ls bug fix." [Online]. Available: http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=6124a3842dfa8484b52e067a8ab8105c3875a4f7

[15] "ls bug report." [Online]. Available: https://debbugs.gnu.org/cgi/bugreport.cgi?bug=11453

[16] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, pp. 100–107, 1998.

[17] P. McMinn, "Search-based software test data generation: A survey," *Software Testing Verification and Reliability (STVR)*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[18] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[19] Michal Zalewski, "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt.

[20] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: Testing for divergences between software versions," in *Proc. of the 38th International Conference on Software Engineering (ICSE'16)*, May 2016.