City Research Online

City, University of London Institutional Repository

Citation: Brain, M. & Malkawi, M. (2024). Misconceptions about Loops in C. In: UNSPECIFIED (pp. 60-66). ACM. ISBN 9798400706219 doi: 10.1145/3652588.3663324

This is the published version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/33426/

Link to published version: https://doi.org/10.1145/3652588.3663324

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

Misconceptions about Loops in C

Martin Brain
City, University of London
London, United Kingdom
martin.brain@city.ac.uk

Mahdi Malkawi
City, University of London
London, United Kingdom
mahdi.malkawi@city.ac.uk

Abstract

Loop analysis is a key component of static analysis tools. Unfortunately, there are several rare edge cases. As a tool moves from academic prototype to production-ready, obscure cases can and do occur. This results in loop analysis being a key source of late-discovered but significant algorithmic bugs. To avoid these, this paper presents a collection of examples and "folklore" challenges in loop analysis.

CCS Concepts: • **Software and its engineering → Software verification**; **Automated static analysis**.

Keywords: Loop Analysis, Software Verification, Static Analysis

ACM Reference Format:
Martin Brain and Mahdi Malkawi. 2024. Misconceptions about Loops in C. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3652588.3663324

1 Introduction

When developing a new static analysis tool, there is little distance between the developers and users. Often they are the same. Diagnosing and debugging issues with the tool is no more difficult than regular development. The distance between users and developers grows as a tool becomes more successful. Identifying faults requires the users to be willing and able to file bug reports. Reproduction may require non-disclosure agreements or travel to customer premises. Confirming fixes may require a full release and deploy cycle. Bugs that would have taken days to fix in early development now take weeks, highlighting the importance of early bug detection.

Bugs in static analysis tools vary in terms of severity and impact. Crashes early in inputs can normally be isolated relatively easily. Correctness bugs from mistakes in the core algorithms are some of the most challenging to find and fix. The worst of these are cases with several edge cases, and it *seems* like only one needs to be fixed. This can lead to a cycle of bug reports and fixes, consuming time, budget and user goodwill frighteningly quickly.

This paper addresses bugs that arise from tool developers' mistaken beliefs and incorrect assumptions about loops and their structure. Experience with various tools including CBMC [17, 18, 21], SPARK [16], 2LS [4] and goto-analyzer [14, 23] at different stages of development has shown that these are a persistent source of complex and late-detected bugs. Loops that do not meet the developer's assumptions are often fixed with a simple workaround for the specific example, rarely solving the entire problem, leading to additional bugs with precarious fixes and ballooning code complexity.

This paper makes the following contributions:

- Section 2 reviews several definitions of loops using different program representations and gives examples highlighting their differences.
- Section 3 explains mistakes (marked ↯) that the authors and others, have made about the structure of loops. These were identified as root causes of correctness bugs found in several tools, with many leading to severe and time-consuming bugs.
- Section 4 provides recommendations for handling loop analysis, bug prevention and checks to avoid some mistakes outlined in section 3.

2 What Are Loops?

To illustrate our definitions we use the "first" loop in C [11]:

```
while (fahr <= upper) {
  celsius = (5.0/9.0) * (fahr-32.0);
  printf("%4.0f %6.1f\n", fahr, celsius);
  fahr = fahr + step;
}
```

This has all the classical features of a loop [19]:

1. The program first checks the *loop condition*...
2. ...which is the only way out of the loop.
3. If the condition is true, the *loop body* is run.
4. The program jumps back to the top of the loop after the loop body finishes and everything repeats.

Unfortunately none of these is true for all loops.

When a program is represented as a parse tree, a loop is any instance of the loop grammar rules, such as the `while` or `for` rules. In the example, we have a single `while` loop, with a loop condition (`fahr <= upper`) and a loop body.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SOAP '24, June 25, 2024, Copenhagen, Denmark
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663324

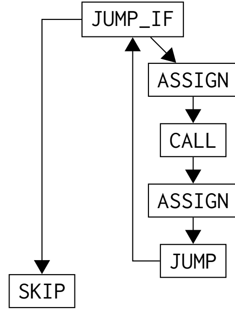

Figure 1. A control flow graph for the first loop example

An alternative representation for the program is a list of instructions. Jumps transfer execution to a different part of the instruction list, dependent on some condition or unconditionally forgoing execution of instructions between the jump and target label. The running example is representable using six instructions:

```
A: JUMP_IF  !(fahr <= upper), B
   ASSIGN   celsius, (5.0/9.0) * (fahr-32.0)
   CALL     printf, "%4.0f %6.1f\n", fahr, celsius
   ASSIGN   fahr, fahr + step
   JUMP     A
B: SKIP
```

The first jump is a conditional, the second is unconditional and is a *backwards jump* as the target is earlier in the list.

A third representation is a control flow graph (CFG) [3], which abstracts the list of instructions into a directed graph. Each instruction becomes a node. Conditional jumps have two edges: to the jump target and to the next instruction. Unconditional jumps have a single edge to their target. All other instructions (except the last) have a single edge to the next instruction. Figure 1 gives the example's CFG.

There are two widely used definitions of loops in CFGs: a natural loop and a cycle. Following [2, 10] we describe a *back edge* as any edge in the graph from $t \longrightarrow h$ where every path from the start of the function to t passes through h. We call h the head of the loop and t the tail of the loop. The corresponding *natural loop* is h plus the set of nodes that can reach t without passing through h. Natural loops with the same head are merged and regarded as a singular loop. Following [9] we describe a *cycle* as a maximal strongly connected component (SCC). Cycles are more general and can contain multiple entry points, unlike natural loops.

Loop Definitions Are Not Equivalent. As all four definitions seek to describe the same fundamental structure, it is not unreasonable to treat them as the same or at least "essentially equivalent". For example, deductive verification tools provide annotations to allow the programmer to state loop invariants. Abstract interpreters and counterexample-guided inductive synthesis based tools also compute loop invariants. Deductive verification tools typically use the parse tree definitions of loops, whilst other tools use the CFG or list definitions. These definition mismatches and differing properties of loops can cause considerable misunderstanding and bugs when trying to combine different kinds of tools [4, 21]. This section presents some sources of these mistakes.

↯ Loop conditions are the same in all definitions. Loop conditions are a feature of the parse tree representation as they are a syntactic feature of most languages. Unfortunately, loop conditions are more fragile than often assumed. In the following code, the parse tree representation has a single loop condition `true` suggesting the loop does not terminate:

```
do {
  open_socket();
  if (connect() == SUCCESS) { break; }
  close_socket();
} while (1);
```

In languages with alternative means of exiting a loop, the syntactic loop conditions are sufficient but not necessary for loop termination . The other routes must be detected and added to precisely reason about the loop exit. See Section 3.3 for more details about the subtleties involved.

The parse tree loop conditions are not directly definable in the instruction lists and CFGs. Compare the following two programs that make use of "shortcut" operators :

```
while (A() && B()) {
  green();
}
```

```
while (A()) {
  if (!B()) { break; }
  green();
}
```

The CFGs are equivalent but the parse tree loop conditions are {`A && B` } and {A} (or {A,B} if augmented as above).

It is tempting to think there is a CFG definition of a loop condition along the lines of "A conditional branch where one branch is outside of the SCC", but this does not match the parse tree definition:

```
while (A() || B() || C()) {
  if (D()) { break; }
  pink();
  if (E()) { break; }
}
```

Here {C, D, E } would be "CFG loop conditions" but {`A || B || C`} (and possibly D and E) would be parse tree loop conditions.

↯ Loop bodies are the same in all definitions. Similarly to loop conditions, loop bodies are a syntactic feature in most languages and are clearly defined. They do not necessarily correspond to what instructions are regarded to be in the loop in other representations. For example, consider:

```
while (choose()) {
  if (choose()) { red(); break; }
  else if (choose()) { goto out; }
}
```

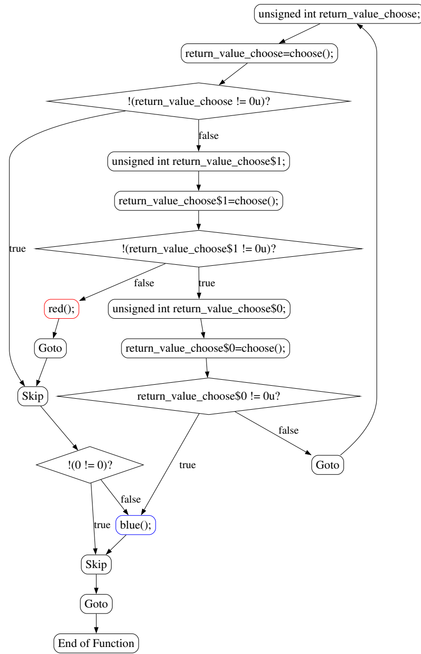

Figure 2. In the CFG, neither `red` or `blue` are in the loop

```
if (0) { out: blue(); }
```

In the parse tree representation, `red` is in the loop body and `blue` is not. The CFG given in Figure 2 shows that neither `red` or `blue` are in the loop as they cannot reach themselves. They are also largely indistinguishable suggesting that the parse tree notion of loop body is not expressible in the CFG.

↯ Back edges are the same as backwards jumps. Backwards edges in a list representation are not guaranteed CFG back edges. Consider the following list representation:

```
    goto A;
B : second();
    assert(counter == 2);
    goto C;
A : first();
    assert(counter == 1);
    goto B;
C : third();
    assert(counter == 3);
    return;
```

In the list representation, `goto B` gives a backwards jump but in the CFG representation it is not a back edge.

↯ The number of loops is the same in all definitions. Compare the following programs:

```
do {
  do {
    blue();
  } while (A());
} while (B());
```

```
do {
  blue();
} while (A() || B());
return;
```

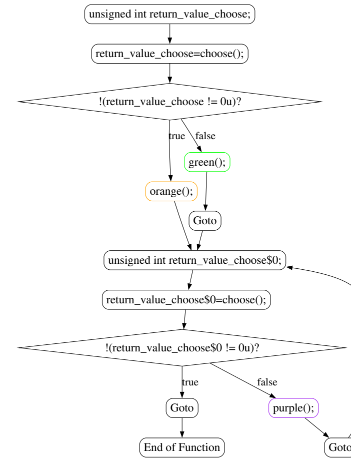

Figure 3. Control flow merges can create multiple loop entry edges

In the parse tree representation, the left example has two loops while the right has only one. However in the CFG representation both have one natural loop.

3 Common Mistakes About Loop Structure

3.1 Entering Loops

↯ Loops have one entry edge. If two or more control flow paths merge at the loop entry and there is no explicit merge node, it is possible to have multiple entry edges. Consider the following program and its CFG given in Figure 3:

```
if (choose())
  green();
else
  orange();

while (choose())
  purple();
```

↯ All entry edges go to the same location. C and some other languages allow writing loops with multiple entry points. Arguably an unintentional "feature" of the language, the labels used by `switch` statements are normal labels and can appear within other control flow structures. Duff [8, 13] used this to provide a manual version of loop unrolling for older compilers and hardware:

```
switch (n & 0x3) {
  do {
    case 0 :  dest[i++] = src[j++];
    case 1 :  dest[i++] = src[j++];
    case 2 :  dest[i++] = src[j++];
    case 3 :  dest[i++] = src[j++];
  } while (j < n);
}
```

Simon Tatham's implementation of coroutines uses a more advanced version of the same idea [22]. Studies suggest cycles with multiple entry points are rare [20].

↯ *Multiple loop entry locations can be fixed by one unrolling.* As entry locations only affect the first iteration, a tempting solution is to unroll the first iteration of any loop with multiple entry locations. In most cases this is a simple and reasonably efficient solution. However [6, 15] shows there are pathological cases which result in an exponential size increase.

↯ *The first instruction must be an entry location.* It is possible for no entry to exist to the obvious first instruction, and that the entry can be from inside a nested loop.

```
if (choose())       goto one;
else if (choose())  goto two;
else                goto three;

while (choose()) {
  while (choose()) {
          red();
  one:    orange();
  two:    yellow();
  three:  green();
  }
}
```

3.2 Inside Loops

↯ *Loops have contents.* In all representations, it is possible to create an empty loop. For example busy-wait loops for low latency inter-thread communication:

```
void busy_wait (void) {
  while (zero_to_unlock);
  return;
}
```

SV-COMP[1] and others [7] use empty loops as an idiom to terminate an analysis path.

```
loop : goto loop;
```

↯ *The entry location is a test for exiting the loop.* C and many other languages have a syntactic construct where this is not true; the `do`/`while` loop:

```
do {
  blue();
} while (choose());
```

It is tempting to think that marking `do`/`while` loops or unrolling the loop once will make this belief true. There are a number of less obvious cases where this does not hold. Depending on whether function calls, pointer validity checks, modifications of variables, etc. need separate instructions this may not start with a conditional exit:

```
while (f00(*(pointer++)) == value)
  yellow();
```

This is also a case where inlining can cause significant changes to the properties of the loop, see Section 3.4.

↯ *End of a loop is an unconditional jump.* Only the parse tree and list representations primarily have a concept of a last instruction. In the list representation, the last instruction must be a backwards jump for it to be a loop, but it can be conditional, for example in the `do`/`while` loop.

↯ *There is a single backwards jump or back edge.* Within the list representation, this depends on whether `continue` is implemented as a jump to the end or as a jump to the start:

```
while (choose()) {
  pink();
  if (choose()) { continue; }
  blue();
}
```

Some languages have control flow statements for "redo this loop iteration (without testing the loop condition)". Perl and Ruby have the `redo` keyword, useful for iterating over data structures or streams where there may not be a way of undoing the loop counter update. The simplest implementation of these is using additional back edges.

↯ *Loops may have multiple back edges but they all go to the same place.* In nested loops the back edge of the inner loop will appear as a back edge into the middle of the outer loop.

```
while (choose()) {
  while (choose()) {
    yellow();
  }
}
```

3.3 Exiting Loops

↯ *Loops have an exit.* Infinite loops can and do happen in correct code. Event driven systems and some control systems will often have one main control loop without an exit.

```
do {
  handle_request();
} while (1);
```

↯ *Loops have a single exit edge.* The `break` gives a simple way to have multiple exit edges to the same location:

```
while (choose()) {
  orange();
  if (choose()) { break; }
  pink();
}
```

↯ *All `break` statements go to the same location.* A subtle consequence of the example in Figure 2 is that the CFG loop exits as soon as a `break` statement is unavoidable, not at the `break`. The instructions between the `if` and `break` statements are the actual exit locations. The following loop

has three exit locations, one for each `break` and one for the loop condition:

```
for (int i = 0; i < firewall->rule_count; ++i) {
 if (firewall->rule[i].matches(packet)) {
  if (firewall->rule[i].type == WHITELIST) {
   packet.status = ACCEPTED; // 1st exit location
   break;
  }
  else if (firewall->rule[i].type == BLACKLIST) {
   packet.status = REJECTED; // 2nd exit location
   break;
  }
 }
}
// 3rd exit location
```

In effect this inserts code "after" the loop exit when using `break` but not when exiting via the loop condition. Python supports `else` statements attached to loops to resolve this asymmetry. Another version of this problem occurs when the loop contains multiple variable declaration scopes and the source or intermediate language requires actions at the end of scope (marking dead variables, C++ destructors, etc.). A `break` may need to perform end of scope actions while the loop condition does not as it is before the start of the scope.

***↯ Loops can only exit using the loop condition or* `break`.** A `return` statement provides a third kind of exit location that is not the same as the loop condition or `break`:

```
while (choose()) {
  orange();
  if (choose()) { return; }
  pink();
}
```

***↯ Loops can only exit using the loop condition,* `break` *or* `return`.** Many languages allow `break` statements to specify the depth of nested loops they are exiting. Kosaraju [12] shows this to be vital to prove a version of the structured programming theorem that does not require additional variables. Thus it can be argued to be a fundamental control flow statement. C does not have break to label so this is regarded as one of the legitimate uses of `goto`, for example:

```
for(unsigned int i = 0; i < WIDTH; ++i) {
  for(unsigned int j = 0; j < HEIGHT; ++j) {
    for(unsigned int k = 0; k < DEPTH; ++k) {
      if (next_cell(i, j, k)) { continue; }
      else if (next_column(i, j, k)) { break; }
      else if (next_row(i, j, k)) { goto nextRow;}
      else process_cell(i, j, k);
      }
    }
nextRow:;
}
```

Another legitimate use of `goto` is the creation of multiple return paths with different resource deallocation and error handling strategies. This is a common pattern in the Linux kernel and can give additional exit locations. The following example provides a degree of robustness against programming errors as unhandled branches and conditions will default to the error exit path:

```
struct subsys *s = kalloc(sizeof(struct subsys));
int err = subsys_init(s, parameters, policy);
if (err) { goto fail_subsys; }

for (unsigned int i = 0; i < s->hook_count; ++i) {
  err = subsys_register_hook(s, i);
  if (err) { goto fail_hook; }
}

if (subsytem_status(s) == FUNCTIONAL)
  { goto success; }

fail_hook:    subsys_unregister_hooks();
fail_subsys: kfree(s);
  klog("Failed to configure subsys %d", err);
  return false;

success:
  klog("Subsys %s configured with %d hooks",
       s->identifier, s->hook_count);
  return true;
```

Some languages have additional control flow statements which provide additional different exit locations for example exception handlers.

3.4 Simplification, Preprocessing and Optimisation

↯ Simplifications do not affect loop analysis. Semantic reasoning can alter the results of loop analysis. For example, a common pattern to force a macro to be used like a statement requires a `do`/`while` loop to "swallow" the semi-colon:

```
#define INIT_SUBSYSTEM(X) do { \
  bzero((X), sizeof(struct subsystem)) ; \
  load_system_config((X)); \
  register_subsystem((X)); \
} while (0)

INIT_SUBSYSTEM(networking);
```

Whether this is regarded as a loop depends on the kind and amount of simplification before the loop analysis.

↯ Simplification only affects control edges decisions. The previous example can be handled during the construction of the CFG by omitting edges that cannot be taken. Unfortunately not all simplifications are so straight-forward:

```
#define POINTER_RW(p) do { *(p) = *(p); } while(0)

do {
  POINTER_RW(p);
  do {
    some_code(p);
```

```
  } while (condition1());
  p = p->next;
} while (p != null);
```

If p is always an accessible address, then POINTER_RW can be removed such that there is one CFG loop rather than two. The converse effect is also possible; adding instructions or instrumentation at the start of loop bodies can increase the number of loops.

↯ Preprocessing can alter the number of loops but not create them from nothing. Depending on the language's requirements and where in the compilation process loop analysis is performed, this may not be true. If the language requires that tail recursive functions are rewritten it is possible for loops to "appear" in an acyclic (but recursive) function.

↯ Inlining is harmless. Inlining allows simulation of context-aware analysis to potentially simplify the handling of small utility functions, and "syntactic sugar". On inlining, `return` statements act as a lightly restricted form of forward `goto` statements. The `return` is replaced with a jump to the calling context. This can give a loop exits which are a significant distance from the loop, may or may not merge with flow control from the loop and are not easily recognised as a special case in the way normal `return` statements are. If the tool performs inlining then a lot of developer intuitions, "This works as long as the input programs do not have `goto`" no longer hold. Many uses of `goto` in this paper can be simulated with inlined `return` statements[1].

↯ Jump threading is harmless. If a jump instruction J targets a second, unconditional jump K, then J can be redirected to directly jump to the target of K. In most cases this simplifies the resultant CFG but it can interact in unexpected ways with other preprocessing steps and undo other simplifications. For example, if `continue` is implemented as a jump to the end of the loop body (see Section 3.2) then jump threading can covert these to additional back edges. If the instruction after a function call is a jump then inlining and then jump threading can create a `goto` from anywhere in the callee to anywhere in the caller.

4 Recommendations

Faced with some of the oddities and edge cases described in this paper, a reasonable reaction is to ask how common they are. In our experience, loop edge cases are more common than anticipated in user-supplied code. Sources include code compiled from languages with non-C control flow constructs like Python's `for-else`, Ada's `loop` or nested `break` instructions, autogenerated code from DSLs (like Simulink), combinations of preprocessing steps, such as inline, loop acceleration, unrolling then simplification, legacy code, hand optimised high-performance code, decompiled optimised code, particularly loop unrolling, splitting, fusion and i-cache layout optimisations and decompiled obfuscated code.

As a tool becomes prosperous, the chances of encountering one or more edge cases rise dramatically. Thus we recommend the following steps:

1. Design for the most general case. For example, if you are using the CFG definitions of a loop, the API should return the set of entry locations and the (possibly empty) set of exit locations. It is fine to have more specific calls such as getting *the* entry location or *the* exit condition but these should have at least a run-time check that the loop is in the required form.
2. Make explicit which cases are not handled. It is reasonable to initially not handle irreducible loops or to handle them with a naïve exponential algorithm. However this should be explicitly checked at run time and documented to the user. Extending support by adding extra cases is strongly preferable to having a complete algorithm for some loops and adding rewrite steps before and after to expand the set of supported loops.
3. If your system makes use of more than one representation then handling the translation between them needs to be part of the design. For example, a system that generates source/parse tree loop invariants using a CFG-based technique will need to track the correspondence between the two definitions of loop. Annotating parse tree concepts such as the loop body and loop conditions into the CFG is one approach but care needs to be taken to preserve them (see Section 3.4).
4. Loop analysis code needs careful and systematic testing with awareness of the possible edge cases. The examples given in this paper are available as a test suite [5]. Each program is a minimal test for the specific issue, with a single input that controls the path taken through the program, providing a single output that records the branches taken. This can be used for back-to-back testing with compilers, interpreters, source-to-source translation, or dynamic analysis tools. If the input/output relation changes, the transformation is buggy and is either inserting or omitting edges. To test static analysis tools, the programs also contain assertions that are only true for valid paths through the program. If an assertion fails, it indicates a bug in the static analysis tool. We provide a usable set-up to test the examples supplied with CBMC.

Acknowledgments

Research reported in this publication was supported by an Amazon Research Award, Fall 2022 CFP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of Amazon.

[1] `returns` should be thought of as domesticated `gotos` but domesticated in the same way that cats are domesticated.

The examples in this paper have been collated over a long period, from bug reports, folklore, edge cases found during development, etc. The authors wish to thank everyone who has had input. Where possible, citations are provided. In particular, Martin Brain would like to thank Peter Schrammel for his development model, Holly Nyx for editorial support, and John Galea for being the person saying "But Martin, that doesn't work because...".

References

[1] 2023. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/

[2] Alfred Vaino Aho, Monica Sin-Ling Lam, Ravi Sethi, and Jeffrey David Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[3] Frances Elizabeth Allen. 1970. Control Flow Analysis. 5, 7 (1970). https://doi.org/10.1145/390013.808479

[4] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by k-Invariants and k-Induction. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 145–161.

[5] Martin Brain and Mahdi Malkawi. 2024. [artifact] Misconceptions About Loops in C. https://doi.org/10.5281/zenodo.11113582

[6] Larry Carter, Jeanne Ferrante, and Clark Thomborson. 2003. Folklore Confirmed: Reducible Flow Graphs Are Exponentially Larger. *SIGPLAN Not.* 38, 1 (jan 2003), 106–114. https://doi.org/10.1145/640128.604141

[7] Byron Cook, Björn Döbel, Daniel Kroening, Norbert Manthey, Martin Pohlack, Elizabeth Polgreen, Michael Tautschnig, and Pawel Wieczorkiewicz. 2020. Using model checking tools to triage the severity of security bugs in the Xen hypervisor. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 185–193. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_26

[8] Tom Duff. 1988. Re: Explanation, please! Usenet. http://doc.cat-v.org/bell_labs/duffs_device

[9] Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (jul 1997), 557–567. https://doi.org/10.1145/262004.262005

[10] Matthew Sterling Hecht and Jeffrey David Ullman. 1974. Characterizations of Reducible Flow Graphs. *J. ACM* 21, 3 (jul 1974), 367–375. https://doi.org/10.1145/321832.321835

[11] Brian Wilson Kernighan and Dennis MacAlistair Ritchie. 1978. *The C Programming Language*. Bell Telephone Laboratories Incorporated.

[12] Sambasiva Rao Kosaraju. 1974. Analysis of structured programs. *J. Comput. System Sci.* 9, 3 (1974), 232–255. https://doi.org/10.1016/S0022-0000(74)80043-7

[13] Chloé Lourseyre. 2021. Duff's device in 2021. https://belaycpp.com/2021/11/18/duffs-device-in-2021/

[14] Daniel Neville, Andrew Malton, Martin Brain, and Daniel Kroening. 2016. Towards automated bounded model checking of API implementations. *CEUR Workshop Proceedings* 1639, 31–42.

[15] Carl D. Offner. 2013. *Notes on graph algorithms used in optimizing compilers*. Technical Report. University of Massachusetts Boston. https://www.cs.umb.edu/~offner/files/flow_graph.pdf

[16] Florian Schanda and Martin Brain. 2012. Using Answer Set Programming in the Development of Verified Software. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 17)*, Agostino Dovier and Vítor Santos Costa (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 72–85. https://doi.org/10.4230/LIPIcs.ICLP.2012.72

[17] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. 2015. Successful Use of Incremental BMC in the Automotive Industry. In *Formal Methods for Industrial Critical Systems*, Manuel Núñez and Matthias Güdemann (Eds.). Springer International Publishing, Cham, 62–77.

[18] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. 2017. Incremental Bounded Model Checking for Embedded Software. *Form. Asp. Comput.* 29, 5 (sep 2017), 911–931. https://doi.org/10.1007/s00165-017-0419-1

[19] Kaitlyn Siu and Marcelo Badari. 2022. *What's A Loop : A Tree House Adventure*. Wayland.

[20] James Stanier and Des Watson. 2012. A Study of Irreducibility in C Programs. *Softw. Pract. Exper.* 42, 1 (jan 2012), 117–130. https://doi.org/10.1002/spe.1059

[21] Youcheng Sun, Martin Brain, Daniel Kroening, Andrew Hawthorn, Thomas Wilson, Florian Schanda, Francisco Javier Guzmán Jiménez, Simon Daniel, Chris Bryan, and Ian Broster. 2017. Functional Requirements-Based Automated Testing for Avionics. In *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*. 170–173. https://doi.org/10.1109/ICECCS.2017.18

[22] Simon Tatham. 2000. Coroutines in C. https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

[23] Tomoya Yamaguchi, Martin Brain, Chirs Ryder, Yosikazu Imai, and Yoshiumi Kawamura. 2019. Application of Abstract Interpretation to the Automotive Electronic Control System. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 425–445.

Received 03-MAR-2024; accepted 2023-04-19; accepted 3 May 2024