20. Oktober 2025

Statische Code-Analyse

Nando Schenk, C6a
Betreut durch Lena Csomor

Maturitätsarbeit 2026
Kantonsschule Zürcher Oberland

Abstract

Speicherbezogene Programmfehler gehören zu den häufigsten und meist ausgenutzten Sicherheitslücken überhaupt. Statische Code-Analyse kann verwendet werden, um genau diese Bugs frühzeitig zu finden. Ziel dieser Arbeit ist es, verschiedene Tools zur statischen Code-Analyse zu untersuchen und zu evaluieren. Hierfür wurden die Tools auf ein anfälliges Programm angewandt und die Resultate anschliessend ausgewertet. Es zeigte sich, dass die falsch-positiv-Rate ziemlich hoch und der Aufwand, gemeldete Fehler auszuwerten, beträchtlich ist. Trotzdem wurden Fehler gefunden. Somit ist die statische Code-Analyse ein wertvolles Werkzeug, um Code sicherer zu machen.

Inhaltsverzeichnis

1 Einleitung

In Sprachen mit manueller Speicherverwaltung sind Programmierfehler, die den Speicher betreffen, weitverbreitet und bilden einen der häufigsten Gründe für Schwachstellen.[1] Solche Schwachstellen reduzieren die Stabilität von IT-Infrastruktur im Allgemeinen und können von Hackern ausgenutzt werden, um potenziell grossen Schaden anzurichten. Mithilfe von Softwareanalyse kann man die Anzahl dieser Bugs reduzieren, jedoch nicht komplett verhindern.[2] Deshalb empfiehlt die NSA, wenn möglich Sprachen mit automatischer Speicherverwaltung zu verwenden, da so viele Schwachstellen gar nicht erst entstehen können.[3]

C und C++, zwei Beispiele für Sprachen mit manueller Speicherverwaltung, gehören jedoch immer noch zu den meistverwendeten Programmiersprachen überhaupt.[4] Daher ist es nicht realistisch, in naher Zukunft grösstenteils moderne Sprachen mit automatischer Speicherverwaltung zu verwenden. Wir müssen uns damit begnügen, bestehenden Code zu verbessern und langsam auszutauschen.

In dieser Arbeit widme ich mich der Softwareanalyse, um genau zu sein der statischen Code-Analyse. Ich untersuche verschiedene Möglichkeiten und Ansätze, um Software statisch zu analysieren, vergleiche diese untereinander und mit anderen Analysemethoden. Der Fokus liegt auf Speicherbugs, darum konzentriere ich mich in dieser Arbeit auf eine Software, die in C geschrieben ist. Ziel ist es, Analyseprogramme auf ein mittelgrosses Softwareprojekt anzuwenden und möglicherweise Softwarefehler zu finden. Daraufhin kann man diese gefundenen Fehler auswerten und so Rückschlüsse auf das jeweilige Analyseprogramm und statische Code-Analyse per se ziehen.

Zuerst bespreche ich das nötige Hintergrundwissen, was statische Code-Analyse überhaupt ist, wofür man sie brauchen kann etc. Daraufhin stelle ich die Methoden für meine Untersuchungen vor, damit diese reproduziert werden können. Im Anschluss folgen die Untersuchungen selbst und zum Schluss werden diese noch ausgewertet, besprochen und in Kontext gestellt. Am Ende der Arbeit findet sich der Anhang und das Literaturverzeichnis.

2 Hintergrund

Dieses Kapitel behandelt die theoretischen Grundlagen zur statischen Code-Analyse, auf denen die folgenden Kapitel aufbauen. Es werden keine eigenen Erkenntnisse präsentiert und keine Schlussfolgerungen gezogen. Grundlegend lässt sich zwischen zwei Formen der Programmanalyse unterscheiden: statische und dynamische Analyse. Bei der dynamischen Code-Analyse wird das zu analysierende Programm ausgeführt und man schaut, wie es unter verschiedenen Konditionen reagiert. Im Gegensatz dazu wird bei der statischen Analyse der Code direkt überprüft.[5]

2.1 Anwendungsbereiche

Statische Code-Analyse findet in vielen Bereichen der Softwareentwicklung Anwendung, vom Linten über die Softwareoptimierung, bis hin zum Finden von ernsten Sicherheitslücken.[6]

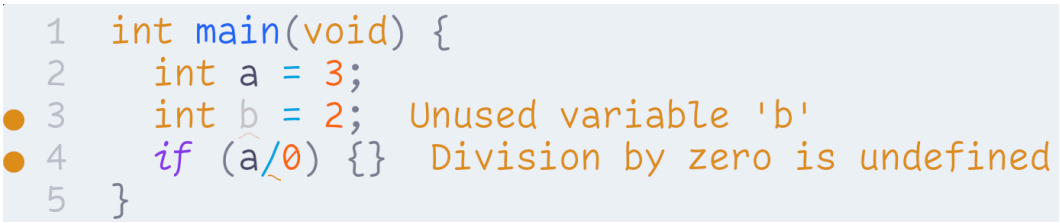


Abbildung 1: Auch eine Form der statischen Code-Analyse: Linting

Die meisten modernen Compiler analysieren Code, um ihn zu optimieren. Clang (und andere Compiler) analysieren Code ebenfalls, um darin Fehler zu finden. Eine einfache Version davon ist das Linten, wie es in Abbildung 1 zu sehen ist. Hier arbeitet im Hintergrund Clang und analysiert das Programm direkt im Editor, um den Programmierer auf Mängel darin hinzuweisen.

2.1.1 Sicherheit

In dieser Arbeit liegt der Fokus auf statischer Code-Analyse zum Finden von Speicherbugs. Speicherbugs entstehen, wenn ein Programm auf Speicherbereiche zugreift (schreiben oder lesen), die ihm nicht zugewiesen wurde. Resultat ist sogenanntes «undefined behaviour», das Programm kann in einem solchen Fall also abstürzen, fehlerhaft weiterlaufen oder – im schlimmsten Fall – gehackt werden. Solche Bugs entstehen häufig durch hängende Zeiger, also Zeiger die eine Adresse auf einen bereits freigegebenen (oder nie reservierten) Speicherbereich besitzen.

2.2 Weitere Begriffe

Es müssen ein paar Begriffe geklärt werden:

2.2.1 Präprozessor

Der Präprozessor nimmt eine Datei als Input, verarbeitet gewisse Direktiven und gibt die erweiterte Datei aus. Der Präprozessor kommt als erster Schritt beim Kompilieren von C-Code zum Einsatz, um unter anderem Bibliotheken einzubinden.

2.2.2 Dereferenzieren

Dereferenzieren bedeutet einer Speicheradresse zu folgen und auf deren Inhalt zuzugreifen.

2.2.3 LLVM IR

Einige Analysetools arbeitet mit der LLVM Intermediate Representation, diese ist ein Zwischenprodukt beim Kompilieren von Source Code mittels Clang.

2.2.4 Attribute

Attribute sind Verhaltensmerkmale, die verschiedenen Codestellen zugewiesen werden können.

2.3 Pfadsensibilität

Bei der statischen Code-Analyse kann man generell zwischen zwei Typen unterscheiden, pfadsensiblen und pfadunsensiblen Tools. Pfadsensible Analysetools berücksichtigen Codeverzweigungen und beschränken sich so auf mögliche Ausführungspfade während pfadunsensible Ansätze alle Pfade als möglich betrachten. Pfadsensible Tools sind somit genauer aber in der Regel rechenintensiver.[7]

3 Methoden

Im Zentrum der Arbeit steht die qualitative und – wo möglich – quantitative Untersuchung von Werkzeugen zur statischen Code-Analyse. Das Programm tmux wurde hierbei als Untersuchungsgegenstand gewählt, aus Gründen, die später in diesem Kapitel dargelegt werden. Insgesamt wurden vier Werkzeuge qualitativ untersucht, von denen drei effektiv auf tmux angewendet werden konnten, namentlich Cppcheck, Clang und GCC. Von diesen wurden Clang und GCC auf die Anzahl, Art und Korrektheit der Fehler überprüft. Ein gemeldeter Codefehler, der aber eigentlich keiner ist, wird im Folgenden als falsch-positiv beschrieben. Clang wurde lediglich auf Anzahl und Art der gefundenen Fehler überprüft. SMOKE konnte aus mehreren Gründen nicht auf tmux angewandt werden.
Die zur Reproduzierbarkeit wichtigen Dateien befinden allesamt auf einem GitHub-Repository. Das Repository beinhaltet diese Arbeit, die Versionen der von mir verwendeten Softwares und alle Befehle, um die Experimente durchzuführen.[8] Im weiteren Verlauf meiner Arbeit wird nicht ständig auf das Repository verwiesen, da es nicht gebraucht wird, um die Arbeit nachzuvollziehen.

3.1 tmux

tmux ist ein 2007 erschienenes C Programm, das seinen Ursprung in OpenBSD[1] hat und auf den meisten unixoiden Betriebssystemen läuft. Es handelt sich um einen Terminalmultiplexer, das heisst man kann mithilfe von tmux in einem Terminal mehrere Tabs, Fenster und Sitzungen laufen lassen und diese auch vom Terminal trennen.[9] Sicherheitsrelevant ist tmux somit nur bedingt, dies erhöht die Chancen Codefehler zu finden. Kritische Software wird in der Tendenz besser getestet. Laut dem Programm `cloc` besteht tmux aus 72′077 Zeilen C Code (inklusive Headerdateien[2], ohne Kommentare etc.). Tmux ist also ein mittel-

[1] auf BSD basierendes Betriebssystem mit Fokus auf Sicherheit
[2] C Code, der Definitionen enthält und in anderen C Code eingebunden wird

grosses, leicht zu erstellendes, nicht sicherheitsrelevantes Programm und eignet sich somit hervorragend als Untersuchungsgegenstand. Da die Funktionsweise von tmux für die Experimente irrelevant ist, wird, wenn Quellcode-Ausschnitte gezeigt werden, auf die Erklärung verzichtet, was dieses Stück Code innerhalb von tmux bewirkt.

4 Statische Code-Analyse

4.1 GCC

GCC ist eine freie[3] Sammlung von Compilern für verschiedene Programmiersprachen und findet eine sehr breite Anwendung auf den verschiedensten Plattformen und Systemen. Seit der 10. Version aus dem Jahre 2020 unterstützt GCC selbst statische Code-Analyse zum Finden von Fehlern.[10] Diese Arbeit wurde von Red Hat finanziert.[11] Da das Analyseprogramm Teil des Compilers ist, ist es mithilfe der bereits bestehenden Infrastruktur implementiert. Hierfür wird ein sogenannter «Inter-procedural optimization pass»[12] angewandt, das heisst, das Analyseprogramm arbeitet in der gleichen Phase wie die Optimierungsmechanismen, die für Optimierungen zwischen verschiedenen Routinen verantwortlich sind. Somit hat das Analyseprogramm Zugriff auf den gesamten Aufrufgraph und muss diesen im Gegensatz zu anderen Analysewerkzeugen nicht selbst kreieren, höchstens modifizieren.

4.1.1 Anwendung

Es ist sehr einfach, die statische Analysefunktionen von GCC auszutesten, da sie direkt im Compiler integriert sind und somit keiner weiteren Konfiguration erfordern. Um den Code mittels GCC statisch zu untersuchen, reicht es, die Option `-fanalyzer` für GCC zu aktivieren. Bei Tmux geht dies durch die Modifikation der Datei `Makefile.am`, indem man die Variable `AM_CFLAGS` um die Option erweitert. Danach kann man so wie im GitHub Repository von Tmux beschrieben[13] den Code kompilieren.

Beim Erstellungsprozess wird sehr viel Text ausgegeben, da jedoch die Warnungen über einen anderen Datenstrom kommen, kann man diese leicht herausfiltern, zum Beispiel in Bash wie folgt: `make 2> warnings.txt`. Zur Auswertung finden sich so alle wichtigen Informationen in der `warnings.txt` Datei.

Der Entwickler, David Malcolm, strebt eine doppelt so lange Kompilierungszeit im Vergleich zur Kompilierung ohne statische Analyse an, was im Verhältnis zu anderen Analysetools sehr sportlich ist. Um Tmux ohne Modifikationen zu Erstellen, benötigte ich 22,98 Sekunden, zum Erstellen inklusive der Analyse 71,52 Sekunden, was über dreifach so lange ist. Er schreibt, dass diese Zeit bei einigen Projekten eingehalten werden könne.[10] Es ist möglich, für die

[3] Frei im Sinne von Freiheit

Kompilierung alle Prozessorkerne zu nutzen, dann wird die relative Zeitdifferenz noch grösser, nämlich 22,344 Sekunden zu 4,145 Sekunden, beziehungsweise über fünfmal so lange.

4.1.2 Auswertung

Insgesamt hat GCC mit dem `-fanalyzer` Argument bei der Kompilation von Tmux 25 zusätzliche Warnungen ausgegeben. Diese lassen sich wie folgt aufteilen:

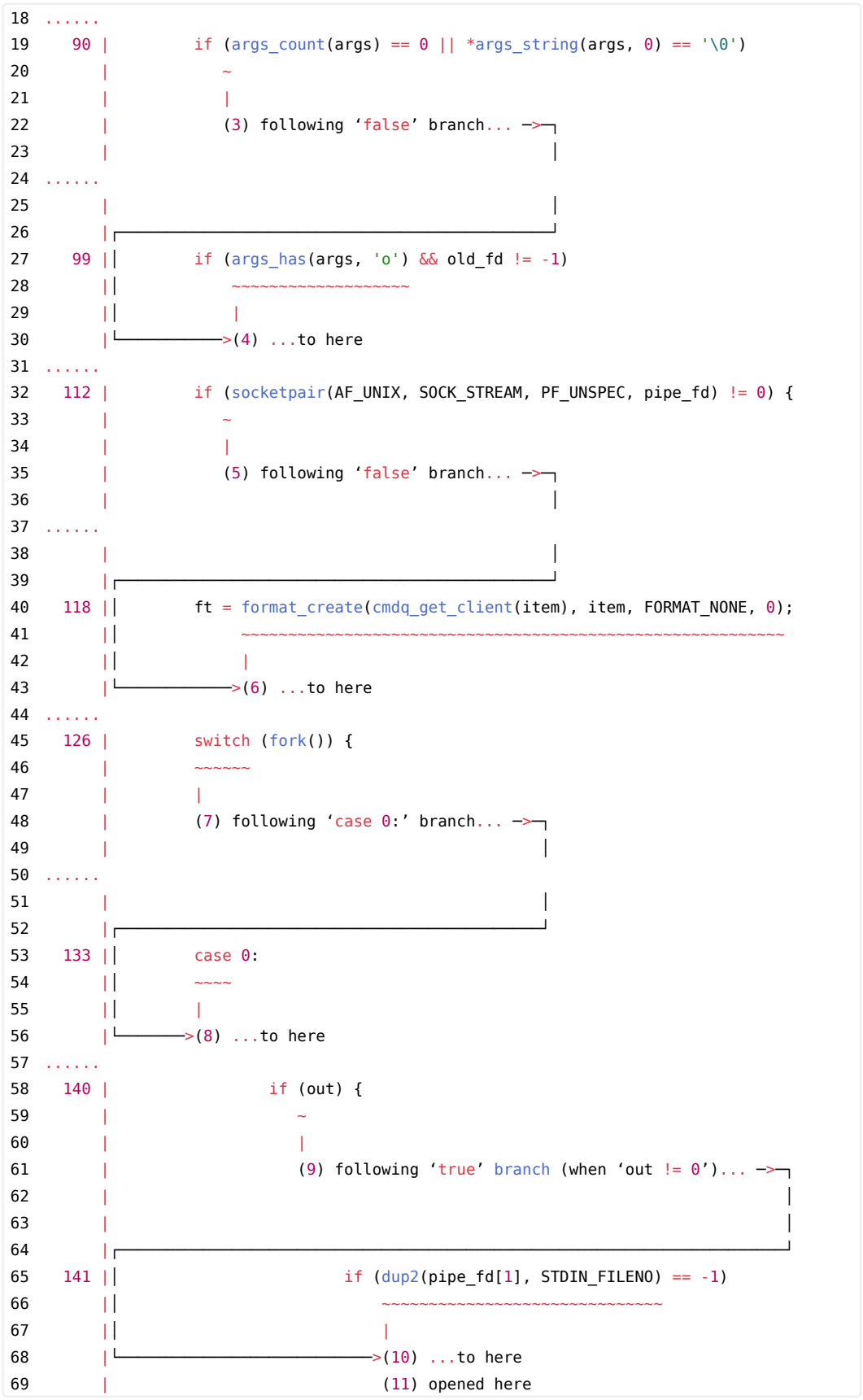
Warnung	**Anzahl**	**Genauigkeit**[4]
-Wanalyzer-use-of-uninitialized-value	3	50%
-Wanalyzer-fd-leak	6	0%
-Wanalyzer-fd-use-without-check	3	0%
-Wanalyzer-null-dereference	10	40%
-Wanalyzer-use-after-free	1	100%
-Wanalyzer-null-argument	2	0%
gesamt	25	25%

Tabelle 1: Auswertung

Eine vollständige, nummerierte Liste der Warnungen findet sich im Anhang in der Tabelle 4. Um zu überprüfen, ob es sich bei einer gegebenen Warnung um einen tatsächlichen Codefehler oder um eine falsch-positive Warnung handelt, muss man den Source Code manuell mithilfe der Informationen des Compilers untersuchen. Exemplarisch wird hier der gesamte Output der Warnung 2 gezeigt:

```
cmd-pipe-pane.c: In function ‘cmd_pipe_pane_exec’:
cmd-pipe-pane.c:141:28: warning: leak of file descriptor ‘dup2(pipe_fd[1],
0)’ [CWE-775] [-Wanalyzer-fd-leak]
  141 |                         if (dup2(pipe_fd[1], STDIN_FILENO) == -1)
      |                            ^
‘cmd_pipe_pane_exec’: events 1-11
   71 |         if (window_pane_exited(wp)) {
      |            ^
      |            |
      |            (1) following ‘false’ branch... ->┐
      |                                              |
......                                               |
      |                                              |
      |┌─────────────────────────────────────────────┘
   77 ||        old_fd = wp->pipe_fd;
      ||        ~~~~~~~~~~~~~~~~~~~~
      ||                 |
      |└────────────────>(2) ...to here
```

[4]gleichwertige Warnungen werden einfach gezählt


18 ......
19 90 | if (args_count(args) == 0 || *args_string(args, 0) == '\0')
20 | ~
21 | |
22 | (3) following ‘false’ branch... ->┐
23 | |
24 ......
25 | |
26 |┌──┘
27 99 || if (args_has(args, 'o') && old_fd != -1)
28 || ~~~~~~~~~~~~~~~~~~~
29 || |
30 |└─────────────>(4) ...to here
31 ......
32 112 | if (socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC, pipe_fd) != 0) {
33 | ~
34 | |
35 | (5) following ‘false’ branch... ->┐
36 | |
37 ......
38 | |
39 |┌──┘
40 118 || ft = format_create(cmdq_get_client(item), item, FORMAT_NONE, 0);
41 || ~~~~~~
42 || |
43 |└─────────────>(6) ...to here
44 ......
45 126 | switch (fork()) {
46 | ~~~~~~
47 | |
48 | (7) following ‘case 0:’ branch... ->┐
49 | |
50 ......
51 | |
52 |┌──┘
53 133 || case 0:
54 || ~~~~
55 || |
56 |└──────>(8) ...to here
57 ......
58 140 | if (out) {
59 | ~
60 | |
61 | (9) following ‘true’ branch (when ‘out != 0’)... ->┐
62 | |
63 | |
64 |┌──┘
65 141 || if (dup2(pipe_fd[1], STDIN_FILENO) == -1)
66 || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
67 || |
68 |└──────────────────────────>(10) ...to here
69 | (11) opened here

```
70   ‘cmd_pipe_pane_exec’: event 12
71   141 |                          if (dup2(pipe_fd[1], STDIN_FILENO) == -1)
72       |                             ^
73       |                             |
74       |                             (12) ⚠ ‘dup2(pipe_fd[1], 0)’ leaks here; was
     opened at (11)
```

Listing 1: Output einer Compilerwarnung von GCC

Zu sehen ist ein Codeverlauf, den GCC entdeckt hat, bei welchem das gefundene Problem auftritt. Besonders wichtig sind die zwei markierten Zeilen, dort wird beschrieben, um was für eine Kategorie von Problem es sich handelt, was das konkrete Problem ist und wo es sich befindet. Dazwischen wird gezeigt, wie dieses Problem zustande kommt, was es verhältnismässig einfach macht, dieses zu überprüfen.
Diese Warnung ist von der Kategorie `-Wanalyzer-fd-leak`, das heisst, GCC denkt, dass hier ein Dateideskriptor reserviert aber nicht, bzw. zu spät freigelassen wird. In diesem Fall stimmt dies nicht:

```c
133 case 0:
134   /* Child process. */
135   proc_clear_signals(server_proc, 1);
136   sigprocmask(SIG_SETMASK, &oldset, NULL);
137   close(pipe_fd[0]);
138
139   null_fd = open(_PATH_DEVNULL, O_WRONLY);
140   if (out) {
141     if (dup2(pipe_fd[1], STDIN_FILENO) == -1)
142       _exit(1);
143   } else {
144     if (dup2(null_fd, STDIN_FILENO) == -1)
145       _exit(1);
146   }
147   if (in) {
148     if (dup2(pipe_fd[1], STDOUT_FILENO) == -1)
149       _exit(1);
150     if (pipe_fd[1] != STDOUT_FILENO)
151       close(pipe_fd[1]);
152   } else {
153     if (dup2(null_fd, STDOUT_FILENO) == -1)
154       _exit(1);
155   }
156   if (dup2(null_fd, STDERR_FILENO) == -1)
157     _exit(1);
158   closefrom(STDERR_FILENO + 1);
159
160   execl(_PATH_BSHELL, "sh", "-c", cmd, (char *) NULL);
161   _exit(1);
```

Listing 2: Problematische Stelle `cmd-pipe-pane.c`

GCC sagt uns (Listing 1), dass bei der Zeile 141 (gelb markiert) der Dateideskriptor `dup2(pipe_fd[1], 0)` entweicht. Die `dup2` Funktion lässt den als zweites Argument übergegebenen Dateideskriptor auf die gleiche Datei zeigen wie der erste. Sie gibt wirklich einen Dateideskriptor zurück, dessen Wert nirgendwo im Code gespeichert wird, was einem Dateideskriptor-Leck entsprechen würde. Jedoch findet sich in der Dokumentation zu `dup2` folgende Bemerkung: *«The dup2() function shall cause the file descriptor fildes2 to refer to the same open file description as the file descriptor fildes […] and shall return fildes2.»*[14] Das bedeutet, `STDIN_FILENO` hat denselben Wert wie der angeblich entwichene Dateideskriptor. Somit hat es hier kein Leck und die Warnung ist falsch-positiv. Tatsächlich kritisieren könnte man, dass `pipe_fd[0]` nicht geschlossen wird, bevor `excl()` aufgerufen wird. So bleibt, während beliebiger Code ausgeführt wird, `pipe_fd[0]` ungenutzt offen, ohne danach wieder gebraucht zu werden.
Sobald ein `dup2` in einer bedingten Anweisung ohne Zuweisung steht, löst GCC eine Warnung aus. Dieses Problem ist nicht leicht sauber zu beheben, da eine statische Analysesoftware gar keine Kenntnis davon hat, dass der Rückgabewert gleich dem zweiten Argument ist. Man müsste entweder mit Attributen und Hinweisen für den Compiler arbeiten oder aber eine einzige Ausnahme für diese Funktion kreieren. Ersteres bedingt sehr grossen Aufwand und Zweiteres müsste für alle Funktionen wiederholt werden, die ein identisches Verhalten aufweisen. Dies stellt keine schöne Lösung dar. Es gibt fünf weitere falsch-positive Warnungen desselben Typs (Nr.4,9,10,11), die sich ebenfalls auf die `dup2` Funktion beziehen, eine davon (Nr. 4) befindet sich in derselben Datei und ist rot hervorgehoben (Listing 2-148).

Auch die 3 Fehler der Kategorie `-Wanalyzer-fd-use-without-check` (Nr. 3,5,6) geschehen, da GCC das Verhalten der `dup2` Funktion nicht vollständig kennt.

```
1 cmd-pipe-pane.c:144:29: warning: ‘dup2’ on possibly invalid file descriptor
  ‘null_fd’ [-Wanalyzer-fd-use-without-check]
```

Die relevanten Zeilen sind im Code (Listing 2) blau markiert. `null_fd` wird tatsächlich mit einem Dateideskriptor initialisiert, wobei jedoch nicht überprüft wird, ob dies erfolgreich war. Würde man nun `null_fd` auslesen, könnte es sein, dass das Programm unvorhersehbar reagiert, falls `null_fd` ungültig ist. Jedoch steht in der Dokumentation zu `dup2` ebenfalls: *«If fildes is not a valid file descriptor, dup2() shall return −1 and shall not close fildes2»*. Das bedeutet, es ist in Ordnung, wenn ein ungültiger Deskriptor übergeben wird. Ebenfalls überprüft der Code direkt, ob `dup2` erfolgreich war oder nicht.

`-Wanalyzer-use-after-free` wurde nur ein Mal ausgelöst (Nr. 13), hier liegt GCC auch richtig.

```
#13 mode-tree.c:1132:32: warning: use after ‘free’ of ‘mtd’ [CWE-416] [-Wanalyzer-use-after-free]
```

```c
static void
mode_tree_remove_ref(struct mode_tree_data *mtd)
{
  if (--mtd->references == 0)
    free(mtd);
}

static void
mode_tree_display_menu(struct mode_tree_data *mtd, [...])
{
[...]
  if ([...]) {
    mode_tree_remove_ref(mtd);
[...]
  }
}

int
mode_tree_key(struct mode_tree_data *mtd,[...])
{
[...]
  if ([...]) {
[...]
    if ([...]) {
      if ([...])
        mode_tree_display_menu(mtd, [...]);
      if (mtd->preview == MODE_TREE_PREVIEW_OFF)
[...]
    return (0);
  }
```

Listing 3: Vereinfachter Code aus mode-tree.c

Das Problem hier ist, dass `mtd` in Zeile 27 dereferenziert wird, ohne zu wissen, ob `mtd` bereits freigegeben wurde. Bevor `mtd` dereferenziert wird, wird, wenn die Konditionen stimmen, `mode_tree_display_menu()` mit `mtd` als Argument aufgerufen. Innerhalb dieser Funktion wird, ebenfalls, nur wenn gewisse Konditionen stimmen, `mode_tree_remove_ref()` aufgerufen, wodurch letztendlich `mtd` freigegeben wird. Passiert dies, wird in Zeile 27 ein hängender Zeiger dereferenziert, was unvorhergesehene Folgen, wie zum Beispiel ein Applikationsabsturz.
Passieren tut dies in der Praxis kaum sehr häufig, da ziemlich viele Bedingungen zugleich erfüllt sein müssten, möglich ist es trotzdem. Ein einziger solcher Fehler ist realistisch gesehen nicht weiter schlimm, doch je mehr dieser unwahrscheinlichen use-after-free Fehler im Programm sind, desto unstabiler läuft es. Benjamin Steenkamer beurteilt solche Bugs wie folgt: *«The issue is also exacerbated by*

the fact that UAF[5] vulnerabilities don't have to be exploited by an attacker to cause a crash, as one can occur through normal program execution when the vulnerability is present.»[15, S. 19]

Die Warnungen Nr. 24 und 25 stimmen, der Code ist jedoch relativ komplex. Ein vereinfachtes Codebeispiel zeigt, was GCC gefunden hat:

```c
int *functionThatMayReturnZero() {
  time_t currTime;
  time(&currTime);
  if (currTime % 2)
    return NULL;
  return (int *)0x1234;
}

void foobar(int **a, int **b) {
  *a = NULL;
  *b = functionThatMayReturnZero();
  if (*b == NULL) {
    return;
  }
  *a = (int *)0x424242;
}

int main() {
  int *a, *b;
  foobar(&a, &b);
  printf("%d", *a);
}
```

Das Problem ist, dass in der Zeile 21 `a` dereferenziert wird, obwohl `a` `NULL` sein könnte. GCC hat gleich zwei solcher Codestrukturen korrekt identifiziert. Würde man in diesem Codebeispiel Zeile 5 auf einen Wert ungleich `NULL` setzen, würde GCC keine Warnung auslösen, was zeigt, dass GCC über Funktionen hinweg ein relativ gutes Verständnis des Codes hat. Dies liegt daran, dass das Analyseprogramm wie oben bereits erwähnt als «Inter-procedural optimization pass» implementiert ist.

Ebenfalls interessant sind die Warnungen Nr. 15 und 16, denn sie überschneiden sich. Nr. 15 zeigt einen Pfad auf, der zu einer Verwendung einer uninitialisierten Variable führt und Nr. 16 zeigt einen längeren Pfad auf, der den Anfang von Nr. 15 erreicht und ab dort identisch ist. Das ist per se nicht falsch, GCC könnte diese zwei Warnungen jedoch kombinieren und als eine ausgeben.

Die Warnungen 7, 8 und 14 kommen alle aus demselben Grund zu Stande:

[5]=**u**se **a**fter **f**ree

GCC kann nur C Quellcode lesen[16]. Daraus resultiert, dass kein Verhalten von Code analysiert werden kann, der entweder bereits kompiliert wurde oder in einer anderen Sprache geschrieben wurde. Im Falle von 7, 8 und 14 wurden relevante Funktionen bereits kompiliert, da sie Teil einer externen Bibliothek sind. Somit hat GCC keinen Zugriff darauf und muss Annahmen über das Verhalten der Funktionen treffen, die unter Umständen falsch sind. Zum Beispiel kann GCC bei Nr. 14 nicht wissen, dass die `strlen` Funktion nie 0 ausgibt, unter der Bedingung, dass zuvor überprüft wurde, dass der erste Charakter der Zeichenkette, die an `strlen` übergeben wurde, ungleich null ist.

Die Warnungen 22 und 23 haben im Kern dasselbe Problem wie die im letzten Abschnitt beschriebenen Warnungen. GCC denkt, Variablen könnten null sein (bei 22 `item->list` und bei 23 `l`), obwohl dies nicht möglich ist. Die Variablen nehmen nämlich den Rückgabewert der `xreallocarray` Funktion an, die wiederum den Rückgabewert von `reallocarray` ausgibt, ausser diese wäre `NULL`, in diesem Fall wird das Programm beendet. 22 und 23 unterscheiden sich trotzdem vom Rest: Die Fehlermeldung lautet *«use of NULL where non-null expected»* und bezieht sich auf ein mögliches Null-Argument an die externe `qsort` Funktion. Doch wie weiss der Compiler, dass das Argument nicht null sein darf, obwohl wir vorher festgestellt haben, dass das Analyseprogramm keine bereits kompilierten Funktionen analysiert? Die Definition von `qsort`, die der Compiler sieht, schaut so aus:

```c
extern void qsort (void *__base, size_t __nmemb, size_t __size, __compar_fn_t
__compar) __nonnull ((1, 4));
```

Relevant hier ist das `__nonnull` Attribut (um genau zu sein ist es ein Makro, hinter dem sich das nonnull Attribut versteckt). GCC und andere Compiler unterstützen viele Funktionsattribute, die dem jeweiligen Compiler Hinweise über den Code geben. Zu beachten ist hierbei, dass die meisten nicht standardisiert sind.[17] Diese Hinweise kann man einerseits für Optimierungen nutzen, aber auch um die Qualität der Code-Analyse zu verbessern. Ohne das `nonnull` Attribut hätte GCC keine Chance gehabt zu wissen, dass `qsort` keinen Nullpointer akzeptiert. Mithilfe dieser Attribute können Compiler korrekte Annahmen treffen, ohne aufwendig Code zu analysieren, sofern das überhaupt möglich ist. Es gibt ebenfalls das `returns_nonnull` Attribut, hätte man `xreallocarray` mit diesem Versehen, hätte GCC gewusst das hier kein Problem entstehen kann und es hätte keine falsch-positive Warnung gegeben.

Die restlichen Warnungen bringen im Wesentlichen keine neuen Erkenntnisse, die meisten übrigen falsch-positiven Warnungen geschehen, da GCC – zumindest scheint es so – jede Kondition unabhängig von den anderen als wahr oder falsch ansieht, insofern diese genug komplex sind. Das bedeutet, wenn eine

Kondition «a» wahr oder falsch ist, Kondition «b» jedoch als nicht «a» definiert wird, würde GCC im folgenden Pseudocode 4 mögliche Pfade anschauen statt nur zwei:

```c
a = maybeTrue()
b = !a
if (a) {
  something1()
}
if (b) {
  something2()
}
```

GCC würde hier auch die Szenarien in Betracht ziehen, dass `something1` und `something2` beziehungsweise keine der Beiden ausgeführt wird. Hier findet demnach ein Kompromiss zwischen Leistung und Genauigkeit statt.

Insgesamt kann man folgende Punkte festhalten:

- GCC analysiert nur C Quellcode und weiss so nichts über das Verhalten in bereits kompilierten Funktionen
- Oft hat es mehrere falsch-positive Warnungen aus demselben Grund, somit ist es auch einfacher mehrere auf ein Mal in GCC zu beheben
- Hinweise in Form von Attributen werden von GCC genutzt, was die Genauigkeit erhöht, insofern genügend Attribute an den richtigen Stellen gesetzt werden
- Ein sehr grosser Anteil der Warnungen sind falsch-positiv, was einerseits an der Natur von statischer Code-Analyse liegt und ebenfalls an der Reife des tmux Projektes liegen kann
- Es werden insbesondere eher schwer zu erreichende Pfade gefunden, die man mittels dynamischer Analyse kaum findet

4.2 Cppcheck

Cppcheck ist eine eigenständige Software zur statischen Code-Analyse und wird seit über 18 Jahren von Daniel Marjamäki entwickelt. Sie ist in vielen Entwicklungstools bereits integriert oder über ein Plugin integrierbar.[18] Die Funktionsweise wird sehr übersichtlich in einem von ihm verfassten Dokument erklärt.[19] Der zu analysierende Code durchläuft mehrere Phasen, bevor er mithilfe von Regeln überprüft wird. Zuerst wird eine Quelldatei mittels eines Präprozessors konvertiert und danach wird der gesamte Code in einzelne Tokens aufgeteilt. Es wird ein Syntax-Baum generiert mit Tokens als Knoten. Daraufhin wird eine allgemeine Analyse durchgeführt, anhand derer verschiedene sogenannte Regeln Fehler erkennen können. Dabei wird jeweils nur eine Datei überprüft.

4.2.1 Anwendung

Cppcheck lässt sich leicht auf den meisten Systemen kompilieren, da viele Buildsysteme[6] unterstützt werden und das Programm keine Abhängigkeiten hat neben dem Buildsystem und dem Compiler. Bereits vorkompilierte Versionen sind ebenfalls vorhanden. Im Gegensatz zu vielen anderen Analysetools ist Cppcheck bei der Anwendung auf keinerlei externe Abhängigkeiten angewiesen, um Sourcecode zu analysieren, was die Anwendung erheblich erleichtert.[20] Auch ist es so simpel, die Software zu nutzen, nachdem sie nicht mehr entwickelt würde. Um tmux mit Cppcheck zu überprüfen, reicht folgender Befehl im Ordner mit dem Quellcode von tmux aus:

```sh
cppcheck .
```

Um alle Prozessorkerne zu nutzen, alle Pfade zu analysieren, die Warnungen in eine Datei zu schreiben und hierbei noch die Zeit zu messen habe ich den folgenden Shell Befehl angewandt:

```sh
time cppcheck --check-level=exhaustive -j $(nproc) . 2> warnings.txt
```

Der Rechenaufwand ist deutlich grösser als der von GCC, nämlich 9 Minuten und 2 Sekunden ohne Parallelisierung und 2 Minuten und 48 Sekunden mit Parallelisierung. Das heisst, Cppcheck benötigt 7,5-mal mehr Zeit.

4.2.2 Auswertung

Gemeldet wurden insgesamt zwei Fehler und zwei Warnungen, wobei ein Fehler korrekt ist und die beiden Warnungen ebenfalls angebracht sind.

[6] Programm, das die Erstellung einer Software automatisiert

```
cmd-find.c:979:3: error: Address of local auto-variable assigned to a function parameter. [autoVariables]
  fs->current = &current;
  ^
compat/imsg-buffer.c:49:13: error: syntax error [syntaxError]
 TAILQ_HEAD(, ibuf) bufs;
            ^
window-tree.c:450:8: warning: Possible null pointer dereference: l [nullPointer]
 qsort(l, n, sizeof *l, window_tree_cmp_window);
       ^
window-tree.c:443:6: note: Assignment 'l=NULL', assigned value is 0
 l = NULL;
     ^
window-tree.c:445:2: note: Assuming condition is false
 RB_FOREACH(wl, winlinks, &s->windows) {
 ^
window-tree.c:450:8: note: Null pointer dereference
 qsort(l, n, sizeof *l, window_tree_cmp_window);
       ^
window-tree.c:496:8: warning: Possible null pointer dereference: l [nullPointer]
 qsort(l, n, sizeof *l, window_tree_cmp_session);
       ^
window-tree.c:483:6: note: Assignment 'l=NULL', assigned value is 0
 l = NULL;
     ^
window-tree.c:485:2: note: Assuming condition is false
 RB_FOREACH(s, sessions, &sessions) {
 ^
window-tree.c:496:8: note: Null pointer dereference
 qsort(l, n, sizeof *l, window_tree_cmp_session);
       ^
```

Listing 4: Resultate von Cppcheck

Da die Trennung zwischen den verschiedenen Warnungen und Fehlern nicht offensichtlich ist, wurde jeweils die erste Zeile der Fehler rot und der Warnungen blau markiert. Der erste Fehler bei Zeile 1 stimmt:

```c
int cmd_find_target(struct cmd_find_state *fs, [...])
{
  struct cmd_find_state  current;

  [...]
  } else if ([...]) {
    fs->current = &current;
    }
    [...]
    return (0);
}
```

Bei `current` handelt es sich um eine lokale Variable, das heisst sie wird nach Beendigung der Funktion freigegeben. Es wird jedoch eine Referenz von `current`

an `fs->current` übergeben, obwohl `fs` ein Parameter dieser Funktion ist. Das bedeutet, dass die aufrufende Funktion eine Referenz auf eine Variable erhält, die es nicht mehr gibt. Dies kann zur Dereferenzierung eines hängenden Zeigers führen, ähnlich zu Tabelle 4 Nr.13.

Der in der Zeile vier beschriebene Fehler ist falsch-positiv und leicht begründbar: `TAILQ_HEAD` ist ein Makro, also ein Stück Text, welches noch vor der Kompilierung ersetzt wird. Eigentlich würde dies Cppcheck verstehen[19], jedoch findet es die Definition für das Makro nicht. Dies ist der Fall, da sich die Quelldatei in einem Ordner befindet, die Datei mit der Definition jedoch nicht. Trotzdem wird die Datei mit der Definition so eingebunden, als ob sie im gleichen Ordner wäre. Beim Kompilieren funktioniert das, da im Buildsystem der Compiler angewiesen wird, alle Dateien in diesem Ordner mit einzubeziehen, also ob sie in der obersten Ordnerhierarchie wären (`Makefile.am`, Zeile 9). Ändert man in `compat/imsg-buffer.c` die Zeile

```c
1 #include "/compat.h"
```

zu

```c
1 #include "../compat.h"
```

wird die Definition für das Makro erfolgreich gefunden und der Text wird vor der Analyse ersetzt. Somit ist dann die Syntax valid und der Fehler weg. Dies zeigt eine Schwäche von Analysetools ausserhalb des Compilers auf: Entweder, sie unterstützen jedes Buildsystem, was sehr aufwendig wäre, oder aber ihnen fehlt unter Umständen essenzielle Informationen über ein Code-Projekt.

Die beiden Warnungen (Zeilen 7 und 19) behandeln identische Code Strukturen an verschiedenen Stellen im Code. Die zweite Warnung ist identisch zu Tabelle 4 Nr. 23, die schon auf Seite 11 besprochen wurde. Dort wurde sie als falsch-positiv gewertet. Cppcheck hat jedoch im Gegensatz zu GCC deutlich weniger Kontext, namentlich nur eine Quelldatei auf ein Mal (inklusive via Präprozessor eingebundene Dateien). Wie GCC interpretiert auch Cppcheck Attribute, um die Qulität der Analyse zu verbessern (`tools/parse-glibc.py`, Zeile 101). Somit weiss auch Cppcheck, dass `qsort` einen nicht-`NULL`-Pointer als erstes Argument erwartet. Die Warnung wäre, wie bereits festgestellt, korrekt, wenn `xreallocarray` `NULL` zurückgeben könnte. Da Cppcheck jedoch nur die Definition von `xreallocarray` sieht und aus dieser das Verhalten nicht offensichtlich ist, geht Cppcheck logischerweise davon aus, dass `NULL` ein zulässiger Rückgabewert sei. In diesem Falle sind die Warnungen also komplett angebracht, aber schlussendlich trotzdem falsch. Da Cppcheck jedoch (anders als GCC) zwischen «warning» und «error» unterscheidet, ist eine falsch-positive Warnung auch weniger schlimm.

Dies kann man über Cppcheck festhalten:

- Cppcheck meldet im Vergleich zu anderen Analysewerkzeugen weniger Fehler, dafür mit höherer Präzision und im Abtausch mit Rechenzeit
- Da Cppcheck nicht in einem Compiler integriert ist fehlt unter Umständen Kontext, die der Compiler hat
- Da Cppcheck das Buildsystem nicht ausliest, fehlt unter Umständen weiterer wichtiger Kontext
- Auch Cppcheck macht sich Attribute zunutze, was die Qualität der Analyse erhöht

4.3 Weitere Programme

4.3.1 Clang

Clang hat ebenfalls ein statisches Analysetool und nutzt verschiedene «checkers», die den Code auf bestimmte Eigenschaften prüfen. Hierbei profitiert das Analysetool von den bereits für die Clang-Infrastruktur geschriebenen Bibliotheken.

4.3.1.1 Anwendung

Clang bietet den Wrapper «scan-build» an, diesen setzt man vor den Build-Befehl und schon funktioniert alles:

```sh
scan-build make
```

Insgesamt wurde für die Analyse (inklusive Kompilation) 93 Sekunden benötigt bei Verwendung aller Kerne und 7 Minuten und 15 Sekunden mit nur einem. Somit platziert sich Clang zwischen GCC und Cppcheck.

4.3.1.2 Auswertung

Insgesamt wurden 47 Fehler gefunden, also deutlich mehr als GCC und Cppcheck. Sie lassen sich in folgende Kategorien aufteilen:

Warnung	**Anzahl**
Argument with ‹nonnull› attribute passed null	8
Value of ‹errno› could be undefined	1
Assigned value is garbage or undefined	4
Dereference of null pointer	10
Division by zero	1
Function call with invalid argument	3
Garbage return value	1
Result of operation is garbage or undefined	4
Uninitialized argument value	5
Use-after-free	9
Dead assignment	1
gesamt	47

Tabelle 2: Auswertung

Aufgrund der Komplexität der Auswertung von den Warnungen wird hier auf die Unterscheidung von falsch-positiven und korrekten Warnungen verzichtet. Es wurden 2 der 4 Meldungen von Cppcheck gefunden und 13 der 25 von GCC. Die identischen Meldungen sind hier aufgelistet:

Codestelle	**Überlappung**
window-tree.c:450	cppcheck
window-tree.c:496	cppcheck, gcc(23)
window-client.c:189	gcc(22)
window-tree.c:949	gcc(24,25)
spawn.c:181	gcc(18)
server-client.c:2946	gcc(17)
cmd-pipe-pane.c:144	gcc(3)
cmd-pipe-pane.c:156	gcc(6)
cmd-pipe-pane.c:153	gcc(5)
server-client.c:613	gcc(15,16)
job.c:157	gcc(9)
arguments.c:750	gcc(1)
mode-tree.c:1132	gcc(13)

Tabelle 3: Mit GCC oder Cppcheck überlappende Meldungen

Bei GCC steht zusätzlich noch die Nummer der Warnung in Klammern (ersichtlich in Tabelle 4). Es ist durchaus bemerkenswert, dass Clang so viel meldet

und hierbei einen grossen Teil der Warnungen von GCC und Cppcheck ebenfalls findet. Die Meldungen werden bei Clang am übersichtlichsten in Form von einer Website dargestellt.
Falsch-positive Fehler sind rot eingefärbt, korrekte grün. Beachtet man nur diese überlappenden Fehler, kommt Clang auf eine Genauigkeit von leicht über 30%, was sowohl Cppcheck als auch GCC schlägt. Ohne Auswertung von all den anderen Warnungen hat dies jedoch keine Aussagekraft.

4.3.2 SMOKE

SMOKE verspricht einen schnelleren, präziseren und besser skalierbaren Ansatz zur Analyse, insbesondere grosser Mengen Code. Im Vergleich zu den zuvor angeschauten Methoden nutzt Smoke zwei voneinander getrennten Analysen, wobei die erste pfadunsensibel ist, um möglichst schnell viel Code zu analysieren. Da die pfadunsensible Analyse jedoch in der Regel bedeutend ungenauer ist, findet in einem zweiten Schritt eine rechenintensivere pfadsensible Analyse statt. Auch bei sehr grossen Mengen Code kann SMOKE so eine gute Leistung bieten, da der grösste Teil des Codes schon in der ersten, schnelleren Phase rausgefiltert wird.[7]

4.3.2.1 Anwendung

SMOKE arbeitet wie einige andere statische Analysetools ([21],[22],[23]) mit der LLVM IR. Der zu untersuchende Code muss mit der Clang-Toolchain der Version 3.6 kompiliert werden. Um den Bitcode (also die IR) zu extrahieren, kann gllvm[24] gebraucht werden. Die genauen Schritte dazu finden sich auf dem zu dieser Arbeit gehörenden GitHub Repository.[8] SMOKE hat eine Website, auf der beschrieben wird, wie man zu den Resultaten kommt und diese Reproduzieren kann.[25] Alle Resultate, ebenso die bereits in das richtige Format gebrachten Dateien, befinden sich auf einem SSH Server, der nicht mehr erreichbar ist.
Auf der Website ist dennoch möglich, SMOKE selbst herunterzuladen. Versucht man nun, SMOKE auf tmux (im erforderten Bitcode-Format) anzuwenden, kommt eine Fehlermeldung:

```
1  Your licence has expired on 2019-06-10. Please contact our sales for licence renew.
```

Ich habe mir SMOKE mithilfe des Reverse Engineering Programms Ghidra angeschaut und bin auf die relevante Stelle gestossen:

```
112  cVar31 = check_date_licence("MjAxOS0wNi0xMA==");
113  if (cVar31 == '\0') {
114    uVar44 = 0xffffffff;
115    goto LAB_0055f3f6;
116  }
```

Abbildung 2: Pseudocode des Lizenzmechanismus in SMOKE

Seltsam ist, dass weder auf der Website noch in der Arbeit von einer Lizenz die Rede war. Zudem ist die Lizenz fest im Code programmiert, es ist also unter normalen Umständen gar nicht möglich die Lizenz zu einer gültigen zu ändern, wenn man eine hätte. Jedoch besteht die Möglichkeit, die Lizenz zu umgehen, indem man die in Abbildung 2 gezeigte Stelle überschreibt. Eine Anleitung hierzu findet sich ebenfalls in dem Repository.

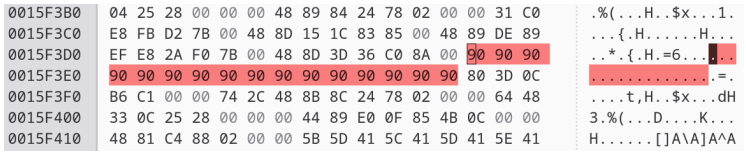


Abbildung 3: Markiert ist die überschriebene Stelle im Programm

Die Beweggründe für das Ablaufdatum sind mir nicht klar. Versucht man nun noch einmal, SMOKE auszuführen, kommt die nächste Fehlermeldung:

```
1 === Glancing Mode : A quick glance of the project, not deep but fast ===
2 error: Invalid value
```

Es wurde jedoch eine Bitcode-Datei in der richtigen Version übergeben, insofern man sich auf die Website und Anleitung im Programm selbst stützen kann:

```
1 USAGE: pp-check [options] <input bitcode>
```

Die Entwickler von SMOKE haben ein Tool namens `pp-capture`, um ein Projekt in das richtige Format zu bringen, dieses befindet sich jedoch auch auf dem Server.

5 Auswertung

Die Auswertung der individuellen Tools wurde bereits in den jeweiligen Kapiteln durchgeführt. Diese hat ergeben, dass durchaus Speicherfehler mithilfe von statischer Code-Analyse gefunden werden können. Einer der grössten Knackpunkte ist, dass oft Annahmen über das Verhalten gewisser Funktionen gemacht werden müssen. Hier sehe ich grosses Potenzial in Attributen, wenn diese erweitert und konsequent eingesetzt würden. Analysewerkzeuge müssten deutlich weniger Arbeit leisten und dabei würde noch die Effizienz und Genauigkeit gesteigert werden. Auch gilt es, den Kompromiss zwischen benötigter Rechenleistung und Genauigkeit zu finden. Je kürzer die Rechenzeit, desto eher wird ein Tool verwendet werden. Jedoch gilt auch: Je unzuverlässiger ein Tool, desto eher wird es gemieden. Am besten werden verschiedene Tools mit unterschiedlichen Stärken und Schwächen kombiniert.[18] Das Beispiel SMOKE hat gezeigt, dass ein Werkzeug noch so toll klingen mag in der Theorie, ohne eine anständige Implementierung und Unterstützung bringt es nichts. Ein weiteres Prachtexemplar hierfür ist K-MELD[21], es wurde jedoch aus Platz- und Zeitgründen weggelassen.

5.1 Vergleich mit anderen Methoden

Dynamische Code-Analyse hat den Vorteil, dass sie auch nicht vorprogrammierte Bugs entdecken kann und weniger Leistung benötigt. Jedoch ist es unwahrscheinlich, mithilfe dynamischer Analyse sehr verschachtelte, abwegige Pfade zu erwischen, die Bugs auslösen können, denn hierfür muss dieser Zustand zufällig im laufenden Programm erreicht werden.

6 Fazit

In dieser Arbeit wurde mithilfe von Experimenten die praktische Anwendung statischer Code-Analyse aufgezeigt. Diese Methode ist bei weitem nicht perfekt, die falsch-positiv-Rate ist je nach Situation sehr hoch und der benötigte Aufwand, um die Warnungen abzuarbeiten ist nicht gering. Trotzdem hat statische Code-Analyse das Potenzial, kritische Bugs zu finden, die anders kaum gefunden werden würden. Der Entwickler von Cppcheck ordnet die Bedeutung der statischen Code-Analyse ganz gut ein: *«No tool covers the whole field. The day when all manual testing will be obsolete because of some tool is very far away.»*[18]

7 Anhang

7.1 GCC Warnungen

Nr.	Korrekt	Warnung
1		arguments.c:750:17: warning: use of uninitialized value ‘error’ [CWE-457] [-Wanalyzer-use-of-uninitialized-value]
2		cmd-pipe-pane.c:141:28: warning: leak of file descriptor ‘dup2(pipe_fd[1], 0)’ [CWE-775] [-Wanalyzer-fd-leak]
3		cmd-pipe-pane.c:144:29: warning: ‘dup2’ on possibly invalid file descriptor ‘null_fd’ [-Wanalyzer-fd-use-without-check]
4		cmd-pipe-pane.c:148:28: warning: leak of file descriptor ‘dup2(pipe_fd[1], 1)’ [CWE-775] [-Wanalyzer-fd-leak]
5		cmd-pipe-pane.c:153:29: warning: ‘dup2’ on possibly invalid file descriptor ‘null_fd’ [-Wanalyzer-fd-use-without-check]
6		cmd-pipe-pane.c:156:21: warning: ‘dup2’ on possibly invalid file descriptor ‘null_fd’ [-Wanalyzer-fd-use-without-check]
7		cmd-rotate-window.c:74:33: warning: dereference of NULL ‘wp’ [CWE-476] [-Wanalyzer-null-dereference]
8		cmd-rotate-window.c:99:33: warning: dereference of NULL ‘wp’ [CWE-476] [-Wanalyzer-null-dereference]
9		job.c:157:28: warning: leak of file descriptor ‘dup2(out[1], 0)’ [CWE-775] [-Wanalyzer-fd-leak]

Nr.	Korrekt	Warnung
10		job.c:160:28: warning: leak of file descriptor ‘dup2(out[1], 1)’ [CWE-775] [-Wanalyzer-fd-leak]
11		job.c:164:36: warning: leak of file descriptor ‘dup2(out[1], 2)’ [CWE-775] [-Wanalyzer-fd-leak]
12		job.c:171:36: warning: leak of file descriptor ‘dup2(nullfd, 2)’ [CWE-775] [-Wanalyzer-fd-leak]
13	x	mode-tree.c:1132:32: warning: use after ‘free’ of ‘mtd’ [CWE-416] [-Wanalyzer-use-after-free]
14		regsub.c:116:18: warning: dereference of NULL ‘0’ [CWE-476] [-Wanalyzer-null-dereference]
15	x	server-client.c:613:52: warning: use of uninitialized value ‘line’ [CWE-457] [-Wanalyzer-use-of-uninitialized-value]
16	Duplikat von 15	server-client.c:613:52: warning: use of uninitialized value ‘line’ [CWE-457] [-Wanalyzer-use-of-uninitialized-value] (bewusst duplikat)
17	x	server-client.c:2946:33: warning: dereference of NULL ‘s’ [CWE-476] [-Wanalyzer-null-dereference]
18		spawn.c:181:23: warning: dereference of NULL ‘w’ [CWE-476] [-Wanalyzer-null-dereference]
19	x	status.c:1691:21: warning: dereference of NULL ‘list’ [CWE-476] [-Wanalyzer-null-dereference]
20		status.c:1907:25: warning: dereference of NULL ‘s’ [CWE-476] [-Wanalyzer-null-dereference]
21		status.c:2062:44: warning: dereference of NULL ‘0’ [CWE-476] [-Wanalyzer-null-dereference]
22		window-client.c:189:9: warning: use of NULL where non-null expected [CWE-476] [-Wanalyzer-null-argument]
23		window-tree.c:496:9: warning: use of NULL ‘l’ where non-null expected [CWE-476] [-Wanalyzer-null-argument]
24	x	window-tree.c:949:22: warning: dereference of NULL ‘other_winlink’ [CWE-476] [-Wanalyzer-null-dereference]
25	x	window-tree.c:949:22: warning: dereference of NULL ‘cur_winlink’ [CWE-476] [-Wanalyzer-null-dereference]

Tabelle 4: Vollständige Liste der durch GCC gefundenen

Bibliografie

[1] Rebert, Alex; Kern, Christoph: *Secure by Design: Google's Perspective on Memory Safety*. Google Security Engineering, 2024.

[2] NSA, CISA: *Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development*. 2025.

[3] NSA: *Software Memory Safety*. 2023.

[4] CISA: *The Case for Memory Safe Roadmaps*. 2023.

[5] Gosain, Anjana; Sharma, Ganga: *Static Analysis: A Survey of Techniques and Tools*. 2015.

[6] Wögerer, Wolfgang: *A Survey of Static Program Analysis Techniques*. 2005.

[7] Gang, Fan et al.: *SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code*. 2019.

[8] dryBoneMarrow: *Maturitätsarbeit 2026 - statische Code-Analyse*. Auf: https://github.com/dryBoneMarrow/Maturaarbeit (abgerufen am 19. Oktober 2025)

[9] Wikipedia: *Tmux*. Auf: https://de.wikipedia.org/w/index.php?title=Tmux&oldid=260646344 (abgerufen am 19. Oktober 2025)

[10] GNU: *StaticAnalyzer - GCC Wiki*. Auf: https://gcc.gnu.org/wiki/StaticAnalyzer (abgerufen am 16. Oktober 2025)

[11] Malcom, David: *Improvements to static analysis in the GCC 13 compiler*. Auf: https://developers.redhat.com/articles/2023/05/31/improvements-static-analysis-gcc-13-compiler (abgerufen am 19. Oktober 2025)

[12] GNU: *IPA passes (GNU Compiler Collection (GCC) Internals)*. Auf: https://gcc.gnu.org/onlinedocs/gccint/IPA-passes.html (abgerufen am 16. Oktober 2025)

[13] OpenBSD: *Installing · tmux/tmux Wiki*. Auf: https://github.com/tmux/tmux/wiki/Installing#from-version-control (abgerufen am 16. Oktober 2025)

[14] Open Group: *dup(3p)*. Auf: https://man.archlinux.org/man/dup.3p.en (abgerufen am 15. Oktober 2025)

[15] Steenkamer, Benjamin P: *An empirical study on use-after-free vulnerabilities*. 2019.

[16] GNU: *Analyzer Internals (GNU Compiler Collection (GCC) Internals)*. Auf: https://gcc.gnu.org/onlinedocs/gccint/Analyzer-Internals.html (abgerufen am 17. Oktober 2025)

[17] cppreference: *Attribute specifier sequence*. Auf: https://en.cppreference.com/w/c/language/attributes.html (abgerufen am 17. Oktober 2025)

[18] Marjamäki, Daniel: *Cppcheck - A tool for static C/C++ code analysis*. Auf: http://cppcheck.net/ (abgerufen am 18. Oktober 2025)

[19] Marjamäki, Daniel: *Cppcheck Design*. 2014.

[20] Marjamäki, Daniel: *danmar/cppcheck*. Auf: https://github.com/danmar/cppcheck (abgerufen am 18. Oktober 2025)

[21] Emamdoost, Navid et al.: *Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning*. 2021

[22] Suzuki, Keita et al.: *Detecting Struct Member-Related Memory Leaks Using Error Code Analysis in Linux Kernel*. 2020.

[23] Wang, Wenwen: *MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernerls*. 2021.

[24] SRI-CSL: *Whole Program LLVM: wllvm ported to go*. Auf: https://github.com/SRI-CSL/gllvm (abgerufen am 19. Oktober 2025)

[25] Gang, Fan et al.: *SMOKE Memory Leak Detector*. Auf: https://smokeml.github.io/ (abgerufen am 19. Oktober 2025)