A Study on Undefined Behavior in C

Vasileios Gemistos
vasilis.gemistos@student.uva.nl

July 19, 2019, 73 pages

Research supervisor:	Dr. Clemens Grelck, c.grelck@uva.nl
Host supervisor:	Dr. Marcel Beemster, marcel@solidsands.com
Host organisation:	Solid Sands, www.solidsands.com

UNIVERSITEIT VAN AMSTERDAM
FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
MASTER SOFTWARE ENGINEERING
http://www.software-engineering-amsterdam.nl

Abstract

In the C programming language, executing an erroneous operation invokes undefined behavior, and anything can happen. The execution of a program that contains code with undefined behavior, will be meaningless. Undefined behavior can cause security vulnerabilities. For example, a signed integer overflow can cause a program to shut down unexpectedly. Undefined behavior may also cause buffer overflow attack, if for example an array out of bounds is not checked. The advantage of undefined behavior in C is that the program avoids the unnecessary checking (such as arrays out of bounds), that results into better performance. The C99 international standard presents a list of 191 undefined behavior cases. In this thesis we select 50 of these cases and we create 62 tests. We classify these 50 test cases into 5 categories: source files, pointers and memory, types, syntax and arithmetic. We write tests for each one of these cases, with increasing complexity and we report the behavior of the compilers GCC, Clang, Intel C++, Tiny C and run-time tool UndefinedBehaviorSanitizer and static analysis tool Frama-C. We present the results for every case, and we document the cases that every compiler, run-time tool and static analysis tool detects. In addition, we present the undefined behavior cases that have been removed or modified from the C99 to the C11 standard. This thesis can be a guide to programmers who want to navigate around undefined behavior in C.

Contents

Chapter 1

Introduction

1.1 What is Undefined Behavior?

In C programs, code can sometimes invoke undefined behavior. The C standard uses the term "Undefined Behavior" as the program behavior, under specific situations that anything can happen. In programming languages such as Java, errors are trapped when they happen. In programming languages such as C and C++, executing an erroneous operation (e.g. division by zero or integer overflow) are said to have Undefined Behavior [Reg10a]. The C99 [C99] language standard, defines the Undefined Behavior as: *"Behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements"*. In C99 there are 191 Undefined Behavior cases that if a program has similar code anything can happen. For example an uninitialized variable or an integer overflow can cause a program to crash, return an undefined result or return the expected result. The latter case is possibly the worst one for a programmer because it is harder to detect it and debug the program. Also, examples have shown us that compilers can give anything as a result output due to undefined behavior [Reg12].

1.2 Why is Undefined Behavior important and what is the problem of undefined behavior?

C is a portable programming language and this is one of its major advantages. The existence of undefined behavior results in better program performance [Lat11a]. The advantage of undefined behavior is that the program will avoid the unnecessary checking (e.g. array out of bounds checking) that would cost time to the program execution. Undefined behavior simplifies the compiler's job, and makes it possible to generate more efficient code in certain situations. This unnecessary checking and compiler's trust that the programmer will not submit code with undefined behavior, can cause a lot of serious problems and vulnerabilities (e.g. security issues, errors and misbehaving programs or abort issues) [WCC$^+$12]. A lot of known open source projects (such as compilers, web browsers, kernels) faced problems due to undefined behavior, like buffer overflow. A major advantage of C is that you can understand the program by simply reading the code. Undefined behavior can take away this advantage [Lat11b, Lat11c, CER].

1.3 Research environment

This research is in collaboration with Solid Sands [1] [SOL]. Solid Sands, based in Amsterdam, is the leading provider of compiler testing and qualification technology in North-America, Europe and Asia. They are improving the quality of C and C++ compilers, libraries and analysis tools. Solid Sands offers SuperTest, a compiler test and validation suite. In this thesis we focus on testing undefined behavior cases from the C99 language standard and document any differences with C11 standard. The C99 standard includes a list of 191 undefined behavior cases. From this list, we select slightly more cases that are decidable by the MISRA catalogue than cases that are not. The MISRA catalogue is a set of guidelines for programmers. In the MISRA catalogue, the committee has made a choice to characterize some undefined behavior cases as decidable and others as not decidable. For example, the division by

[1] https://www.solidsands.com/

zero is not decidable because you have to predict if the divisor can ever be zero when the program runs. Such a prediction is very hard in the general case because according to computer theory it is undecidable to predict the outcome of any program. The MISRA catalogue is an inventory of the undefined behavior cases that the MISRA committee thinks that should be detected. For our selection, we use the MISRA catalogue that Solid Sands provides. We select 50 undefined behavior cases and we create 62 tests. In this thesis we focus on the C99 standard because we want to align with the MISRA catalogue that Solid Sands use. However, we make a report about the differences or modifications of undefined behavior cases from the ISO C99 standard to the ISO C11 standard, mentioned in section 8.1.

1.4 Test process

The test process includes one or more tests with increasing complexity for each undefined behavior example that we test in this research. There are a lot of tools that can detect several cases of undefined behavior. The static analysis tools (such as Frama-C or PC-Lint) can make a report and check the code statically [FRA, PCL]. These tools can help us to detect code with undefined behavior before program execution. Static analysis tools create an analysis report of the given code and can detect undefined behavior. However, run-time tools are very helpful also in detecting such behavior. A run-time tool is UndefinedBehaviorSanitizer (also known as UBSAN) that can be found in compilers GCC and Clang [UBS]. In this research we test the cases of undefined behavior in five compilers, in one static analysis tool and in one run-time tool. After this procedure we report the results into decidable undefined behavior cases and undecidable. The tests that the compilers or the static analysis tool catch the undefined behavior in the code, belong to decidable cases. On the other hand, undecidable are the cases that the run-time tool recognizes or none of the tools and compilers detect. For each case we present the output of every tested compiler and tool in this research. In this study, the test process includes:

- four compilers:
  - GCC [GCC],
  - Clang [Cla],
  - Intel C++ Compiler [ICC],
  - Tiny C Compiler - TCC [TCC],
- one static analysis tool: Frama-C
- one run-time tool: UndefinedBehaviorSanitizer.

For each test file we test the five compilers and two tools that mentioned above. First, without optimization levels and then with optimization levels 1 to 3. Tiny C does not offer optimization levels. For cases that GCC and Clang do not show an error during the execution, we test the run-time tool UndefinedBehaviorSanitizer (UBSAN) also with all optimization levels mentioned above. Also, we test Frama-C static analyzer for each undefined behavior case. In this research we select to use open source compilers with high maintenance level (GCC, Clang) and known industrial compilers (Intel C++). In addition, we include Tiny C that is an open source project as well, which is not maintained as GCC and Clang in order to record the different results. Next to each test case there is the number of undefined behavior from C99 language standard matrix. This matrix can be found in the A. The command line flags that we use are the following:

- -Wall,
- -std=c99,
- -pedantic,
- -O/1/2/3.

In this thesis, we classify the tested cases into 5 categories: source files, pointers and memory, types, syntax and arithmetic. We execute every test under each one of the listed compilers and tools. We report the behavior of each tool and compiler into a table and we make a final verdict for each undefined behavior case. This research will hopefully help Solid Sands and any other interested researcher or professional to find easily the wanted case.

1.5 What will be the use of this thesis?

For all the reasons above, programmers must know in depth the behavior of their program. In order to do that, a guide of how to navigate around undefined behavior cases will be very useful. A programmer will have to make sure that he/she will not submit a program with undefined behavior. If in a program's execution, a part has undefined behavior then this execution will be meaningless. This thesis, will hopefully help a programmer to avoid undefined behavior cases that are listed in this study and to understand why undefined behavior is dangerous for C projects. In addition, this thesis can be a guideline for programmers who want to develop a tool or a compiler for undefined behavior detection. They can consult our work to test their tool or compiler in order to see if it can detects cases that existing compilers and tools are able to recognize.

The following research questions will guide this research:

- How can we classify if an undefined behavior is detectable statically, detectable at run-time or not detectable at all?
- What conclusions can we drawn from the tests of undefined behavior about the compilers and tools?

1.6 Outline

In Chapter 2, we provide important background information about our research and we present simple examples of undefined behavior in C. In Chapters 3 - 7, we present in depth the tests for each undefined behavior case and then we observe the behavior of the tested compilers, run-time tool and static analysis tool. Discussion is described in Chapter 8 and related work in Chapter 9. Conclusions are in Chapter 10. Finally, in Appendix A is the annex J.2 of undefined behavior from the C99 language standard, with all the undefined behavior cases numbered.

Chapter 2

Background

In this chapter we present some basic examples about undefined behavior in C, and a summary of the tools and compilers we use in this study. In addition, we establish the terminology from the C99 standard.

2.1 Terminology from the C99 standard

In this thesis the main terminology used in Chapters 3 - 7 is from the C99 international language standard.

- **Undefined Behavior** (clause 3.4.3): behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.
  NOTE: Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).
- **Object** (clause 3.14): region of data storage in the execution environment, the contents of which can represent.
- **Bit** (clause 3.6): unit of data storage in the execution environment large enough to hold an object that may have one of two values.
- **Byte** (clause 3.6): addressable unit of data storage large enough to hold any member of the basic character set of the execution environment.
- **Behavior** (clause 3.4): external appearance or action.
- **Indeterminate value** (clause 3.17.2): either an unspecified value or a trap representation.
- **Unspecified value** (clause 3.17.3): valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance.
- **Lvalue** (clause 6.3.2.1): an lvalue is an expression with an object type or an incomplete type other than void.

2.2 Examples

The following example is called integer overflow, adding a number to the maximum representable integer invokes undefined behavior.

```c
#include <limits.h>
#include <stdio.h>

int main(void){
    printf("%d\n", (INT_MAX + 1) < 0 );
}
```

Listing 2.1: Integer Overflow

As we can see in the listing 2.1, the program adds one to the biggest representable integer, and then checks if the result is negative or not. For this example the output after the execution could be 1, 0, 52 or a warning. Another common example with undefined behavior is the division by zero.

```c
void fun(int a, int b){
    int c = a/b;
    printf("%d\n", c);
}
```

Listing 2.2: Division by zero

In example 2.2 we can observe that a division by zero can happen if `b` equals zero. In C language this operation invokes undefined behavior, and allows the compiler to handle it differently. It is easy to avoid this error. However, other cases with division by zero is not possible to be detected by compilers and tools. In Chapter 7 we make an analysis for integer overflow and division by zero and how compilers and tools are handling these cases.

2.3 Compilers and tools

For this research three well-known, well-established and well-maintained compilers (GCC, Clang and Intel C++) will be tested. Also, for additional results another compiler (Tiny C) will be tested, as a less famous and not regularly maintained compiler. Many undefined behavior cases can be tested with surprising results on different compilers. Also, a very important aspect of this research is that we use two tools for static and dynamic analysis purposes. The static analyzer tool we use is Frama-C and the run-time tool is UndefinedBehaviorSanitizer (UBSAN) for Clang and GCC.

- **GNU Compiler Collection (GCC)** is a compiler system by GNU project that supports a lot of programming languages. GCC is one of the most known and famous compilers for C programming language. The first release was made in 1987, on May 23, 32 years ago. In this study we use version 8.3.0.
- **Clang** is a compiler system created by Chris Lattner. Clang project provides a language compiler front-end and tooling infrastructure for languages in the C language family for the LLVM project. It is relatively new compiler, the first release was made on September 26 of 2007, 11 years ago. We use version 8.0.0.
- **Intel C++ Compiler** also known as ICC, is a group of C and C++ compilers from Intel for operating systems Windows, Mac, Linux, FreeBSD and intel-based Android devices. In this research we use version 19.0.3.
- **Tiny C Compiler (TCC)** is a compiler created by Fabrice Bellard. This compiler is less known, maintained and mature than the others, the latest version 0.9.27, released 18 months ago.
- **UndefinedBehaviorSanitizer** is a run-time tool for compilers GCC and Clang. This tool can report run-time errors and warnings for several cases of the C programming language.
- **Frama-C** is a static analyzer for C programs by Commissariat à l'Énergie Atomique (CEA-List) and Inria. In this study we use version 18.0 Argon.

Chapter 3

Undefined behavior in source files

In this category of the undefined behavior tests, we include every case relevant to the environment of the language such as translation phases, program structures and hosted environment.

3.1 No new-line character [Number 2]

As mentioned in the C99 Annex J.2 ”A non-empty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment”. Also, in clause 5.1.1.2, C99 say that ”a source file that is not empty shall end in a new-line character.”

```c
#include <stdio.h>

int main(void){ return 0; } /*file does not end with the new-line character.*/
```

Listing 3.1: New-line character missing

Compiler/tool	Warning
GCC/GCC -O0/1/2/3	-
Clang/Clang -O0/1/2/3	no newline at end of file
Intel/Intel -O0/1/2/3	-
TCC	-
GCC UBSAN/GCC UBSAN -O0/1/2/3	-
Clang UBSAN/Clang UBSAN -O0/1/2/3	-
Frama-C	-

Table 3.1: Compilers and tools output for test in 3.1

As we observe GCC does not recognize the undefined behavior and compiles the program until the end without warnings or errors. The same results can be found in Intel C++ and in Tiny C. On the other hand, Clang warns the user that a new-line character is missing from the program. In addition, no tool reports any warning or error related to the undefined behavior.

3.2 Function Main [Number 4]

If a program in a hosted environment does not define a function named `main` using one of the specified forms, then the behavior of the program is undefined. For this case, we write two tests, with function `main` in a non compliant form. In the first test 3.3 the function `main` does not have a specified form as the C standard requires. In clause 5.1.2.2.1 the C99 standard mention that function `main` shall be defined with a return type of `int` and without parameters or with two parameters (`argc` and `argv`), as the code in the listing 3.2 shows.

```c
/*First form, with return type of int and with no parameters.*/
int main(void)  {
/* .... */
}

/*Or with two parameters referred here as argc and argv*/
int main(int argc, char *argv[]){
    /* .... */
}
```

Listing 3.2: Function `main` specified forms

First test

```c
1 #include <stdio.h>
2
3 int main(char ub)  {
4     printf("hello world \n");
5
6     return 0;
7 }
```

Listing 3.3: Test 1: Function `main` without sepcified forms

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	First argument of 'main' should be 'int' 'main' takes only zero or two arguments	hello world
Clang/Clang -O0/1/2/3	error: first parameter of 'main' (argument count) must be of type 'int'	-
Intel/Intel -O0/1/2/3	-	hello world
TCC	-	hello world
GCC UBSAN/UBSAN -O0/1/2/3	-	hello world
Clang UBSAN/UBSAN -O0/1/2/3	-	-
Frama-C	-	-

Table 3.2: Compilers and tools output for test in 3.3

In table 3.2 we can observe that GCC in all optimization levels detects the undefined behavior and shows a warning, but prints out the 'hello world' from line 5 in 3.3. Unlike GCC, Clang constitutes an error. Finally, Intel C++ in all optimization levels and Tiny C execute the program with no warning or error and show the 'hello world'. None of the tools present results, neither Frama-C nor UBSAN recognize the undefined behavior.

Second test

The second test reports similar results as the code in 3.3.

```c
#include <stdio.h>

int main(int ub)  {
    printf("hello world \n");

    return 0;
}
```

Listing 3.4: Test 2: Function Main not in sepcified forms

In the following table are the results of the execution for the second test.

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: 'main' takes only zero or two arguments	hello world
Clang/Clang -O0/1/2/3	warning: only one parameter on 'main' declaration	hello world
Intel/Intel -O0/1/2/3	-	hello world
TCC	-	hello world
GCC UBSAN/UBSAN -O0/1/2/3	-	hello world
Clang UBSAN/UBSAN -O0/1/2/3	-	-
Frama-C	-	-

Table 3.3: Compilers and tools output for test in 3.4

The execution of test 2 reports interesting results. The major difference from the first test in 3.3 is that in Clang the error disappeared and in its place is a warning. That means that the program executes and prints 'hello world'. Intel C++ and Tiny C report the same results as before. In GCC we can observe a small difference, one instead of two warnings. GCC and Clang present differences from test in 3.3. Neither Frama-C nor UBSAN recognize the undefined behavior.

3.3 Non-basic source character [Number 5]

The C99 language standard says that if a character that is not in the basic source characters is encountered in a source file, except in an identifier, character constant, string literal, header name, comment or a preprocessing token that is never converted to a token, the program's behavior is undefined. A simple test of that case is the following example 3.5. An integer takes a value `$3`, which the dollar sign is not a basic source character, so the following code invokes undefined behavior. The basic source characters as C99 mention, are:

- the 26 uppercase letters of the Latin alphabet: A-Z
- the 26 lowercase letters of the Latin alphabet: a-z
- the 10 decimal digits: 0-9
- the following 29 graphic characters: ! ” # % & ’ ( ) * + , - . / : ; < = > ? [\ ] ˆ _ { | } ˜

```c
#include <stdio.h>

int main(){
    int var = $3; /*undefined behavior*/

    return 0;
}
```

Listing 3.5: Non-basic source character

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	error: '$3' undeclared (first use in this function)
Clang/Clang -O0/1/2/3	warning: '$' in identifier error: use of undeclared identifier '$3'
Intel/Intel -O0/1/2/3	error: identifier "$3" is undefined
TCC	error: identifier expected
Frama-C	invalid symbol

Table 3.4: Compilers/tool output for test in 3.5

Table 3.4 shows us that every compiler and tool we use for this test detects the error. We do not use UBSAN for this case because it is a run-time tool and for this test all compilers report an error.

3.4 Identifier with internal and external linkage [Number 7]

The behavior is undefined if for the same identifier there is an external and internal linkage within the same translation unit.

```c
/*First test*/
static int a = 23; /*internal linkage*/

int main(){
    int a; /*no linkage*/
    extern int a; /*external linkage*/

    return 0;
}

/*Second test*/
static int a = 23; /*internal linkage*/

int main(){
    extern int a; /*external linkage*/

    return 0;
}

/*Third test*/
static int a = 23; /*internal linkage*/

int main(){
    extern int a = 3; /*external linkage*/

    return 0;
}

/*Fourth test*/
static int a; /*internal linkage*/

int main(){
    extern int a; /*external linkage*/

    return 0;
}
```

Listing 3.6: Four tests for the identifier with both internal and exernal linkage

First test

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	error: variable previously declared ’static’ redeclared ’extern’ error: extern declaration of ’a’ follows declaration with no linkage
Clang/Clang -O0/1/2/3	error: extern declaration of ’a’ follows non-extern declaration
Intel/Intel -O0/1/2/3	error: ”a” has already been declared in the current scope
TCC	-
Frama-C	-

Table 3.5: Compilers/tool output for the first test from 3.6

GCC, Clang and Intel C++ in all optimization levels detect the undefined behavior and report an error for the identifier a. Tiny C doesn’t report anything and executes the program with no errors or warnings. UBSAN for Clang and GCC could not be used for this case because both compilers report errors. Frama-C doesn’t report anything.

Second test

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	warning: unused variable ’a’ warning: ’a’ defined but not used
Clang/Clang -O0/1/2/3	warning: unused variable ’a’
Intel/Intel -O0/1/2/3	-
TCC	warning: storage mismatch for redefinition of ’a’
GCC UBSAN/UBSAN -O0/1/2/3	-
Clang UBSAN/UBSAN -O0/1/2/3	-
Frama-C	-

Table 3.6: Compilers/tool output for the second test from 3.6

In the second test, no errors occur during the execution of the program by any compiler or tool. GCC, Clang and Tiny C report only warnings about unused variable and storage mismatch. In addition, no tool either UBSAN or Frama-C report anything related to this case.

Third test

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	error: ’a’ has both ’extern’ and initializer warning: unused variable ’a’ warning: ’a’ defined but not used
Clang/Clang -O0/1/2/3	error: ’extern’ variable cannot have an initializer
Intel/Intel -O0/1/2/3	error: an initializer is not allowed on a local declaration of an extern variable
TCC	warning: storage mismatch for redefinition of ’a’ error(line 24 3.6): ’;’ expected (got ”=”)
Frama-C	Global a was already defined

Table 3.7: Compilers/tool output for the third test from 3.6

In the third test, all compilers detect the error of both internal and external linkage in the same translation unit. Frama-C also reports a warning about the case. We do not use UBSAN for this case because it is a run-time tool and for this test GCC and Clang report an error.

Fourth test

Compiler/tool	**Warning/Error**
GCC/GCC -O0/1/2/3	warning: unused variable 'a' warning: 'a' defined but not used
Clang/Clang -O0/1/2/3	warning: unsused variable 'a'
Intel/Intel -O0/1/2/3	-
TCC	warning: storage mismatch for redefinition of 'a'
Frama-C	-
GCC UBSAN/UBSAN -O0/1/2/3	-
Clang UBSAN/UBSAN -O0/1/2/3	-

Table 3.8: Compilers/tool output for the fourth test from 3.6

This test presents the same results as the second test. All compilers except Intel C++, present warnings and no tool recognizes the undefined behavior in this test.

Chapter 4

Undefined behavior in pointers and memory

The undefined behavior cases in this section are related to memory issues, pointers and arrays. Invalid uses of pointer values and array out of bounds access, are some of the examples of this section.

4.1 Value of a pointer to object with ended lifetime [Number 9]

If the value of a pointer to an object whose lifetime has ended is used, then the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

First test

```c
#include <stdio.h>
#include <stdlib.h>

int* fun(int varMain){
    int varFun = 0;
    varFun = varMain + 10;

    return &varFun;
}

int main(){
    int *ptr;

    ptr = fun(10);
    *ptr = *ptr + 1;
    printf("ptr = %d\n", *ptr);

    return 0;
}
```

Listing 4.1: Pointer to object with ended lifetime, test 1

In this test example, function `fun` accepts an integer and returns the address of an object. In function `main` the pointer `ptr` is used to call the function `fun` that returns the address of the object `varFun` whose lifetime ends with the return of the function. Afterwards the use of the pointer `ptr` that points to `varFun`, is undefined behavior.

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: fuction 'fun' returns address of local variable	Seg fault(core dumped)
Clang/Clang -O0	warning: address of stack memory associated with local variable 'varFun' returned	ptr=21
Clang -O1	warning: address of stack memory associated with local variable 'varFun' returned	ptr=32768
Clang -O2	warning: address of stack memory associated with local variable 'varFun' returned	ptr=-672170136
Clang -O3	warning: address of stack memory associated with local variable 'varFun' returned	ptr=-1080871112
Intel/Intel -O0/1/2/3	warning #1251: returning pointer to local variable	ptr=21
TCC	-	ptr=21
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: load of null pointer of type 'int'	Seg Fault(code dumped)
Clang UBSAN/UBSAN -O0	-	ptr=21
Clang UBSAN -O1/2/3	-	ptr=1
Frama-C	warning: locals {varFun} escaping the scope of fun through warning: accessing left-value that contains escaping addresses	-

Table 4.1: Compilers and tools output for test in 4.1

In this test GCC shows a warning that function `fun` returns a local variable address, and the output of the program is 'segmentation fault', unable to print anything. Clang presents the same warning for all optimization levels but different results than GCC. Without optimization, the output is `ptr=21`, but for the optimization levels 1-3 the output values are completely undefined as the table 4.1 shows. Intel C++ prints a warning for returning pointer to local variable and then prints the seemingly expected result, `ptr=21`. Tiny C presents the same result without warnings. UBSAN for GCC reports a run-time error and the output is 'segmentation fault'. UBSAN for Clang does not report any warnings or errors but the output `ptr=1` is odd for optimization levels 1-3. Frama-C detects the undefined behavior for this test file and reports the needed warnings.

Second test

```c
#include <stdio.h>

const char *ptr;

void func(){
    const char c_str[] = "This is it";
    ptr = c_str; /*undefined behavior*/
}

void what_happened(){
    printf("%s\n", ptr);
}

int main(){
    func();
    what_happened();
    return 0;
}
```

Listing 4.2: Pointer to object with ended lifetime, test 2

This example is similar to the non-compliant code that Carnegie Mellon university created [Sea18]. In this code the address of the variable `c_str` is assigned to variable `ptr`. The assignment is valid but the `c_str` cannot go out of the scope of `func` while `ptr` holds its address. After `func` terminates, the function `what_happened` prints the string that `ptr` points to.

Compiler/tool	Warning/Error	Output
GCC/GCC -O0	-	blank
GCC -O1	-	U
GCC -O2/3	-	V
Clang/Clang -O0	-	This is it
Clang -O1/2/3	-	blank
Intel/Intel -O0/2/3	-	This is it
Intel -O1	-	blank
TCC	-	Thi?@
GCC UBSAN/UBSAN -O2/3	-	V
GCC UBSAN -O0/1	-	U
Clang UBSAN/UBSAN -O0	-	This is it
Clang UBSAN -O1/2/3	-	blank
Frama-C	Warning: locals {c_str} escaping the scope of func through ptr	-

Table 4.2: Compilers output for test in 4.2

It is obvious from the table 4.2 that only Frama-C detects the undefined behavior for this case and presents a relevant warning. All other compilers and tools show undefined results without any warning or error. In table 4.2 the blank in the output means that the compiler prints just the line without text.

Third test

```
#include <stdio.h>
#include <stdlib.h>

int *gl_ptr;

void func2(){
    int a = 23;
    gl_ptr = &a;
}

void func1(){
    func2();
}

int main(){
    func1();
    printf("gl_ptr = %d\n", *gl_ptr);
    return 0;
}
```

Listing 4.3: Pointer to object with ended lifetime, test 3

Compiler/tool	Warning/Error	Output
GCC/GCC -O0	-	gl_ptr=23
GCC -O1	warning(line 17): 'a' is used uninitialized in this function	gl_ptr=32764
GCC -O2	warning(line 17): 'a' is used uninitialized in this function	gl_ptr=32765
GCC -O3	warning(line 17): 'a' is used uninitialized in this function	gl_ptr=32766
Clang/Clang -O0	-	gl_ptr=23
Clang -O1	-	gl_ptr=32766
Clang -O2/3	-	gl_ptr=0
Intel/Intel -O0/1/2/3	-	gl_ptr=23
TCC	-	gl_ptr=23
GCC UBSAN/UBSAN -O0	-	gl_ptr=23
GCC UBSAN -O1/3	-	gl_ptr=32766
GCC UBSAN -O2	-	gl_ptr=32765
Clang UBSAN/UBSAN -O0	-	gl_ptr=23
Clang UBSAN -O1	-	gl_ptr=32764
Clang UBSAN -O2/3	-	gl_ptr=0
Frama-C	Warning: locals {a} escaping the scope of func2 through gl_ptr	-

Table 4.3: Compilers and tools for test in 4.3

Frama-C detects the undefined behavior of the third test. All compilers do not present any relevant warning or error and their output is either undefined or the seemingly expected result.

4.2 Use of indeterminate value [Number 10]

The value of an object with automatic storage duration is used while it is indeterminate.

First test

In this test the value of object `num1` which is uninitialized is used in line 5 to initialize object `num2`. This though invokes undefined behavior as stated in the C99 standard.

```
 1  #include <stdio.h>
 2
 3  int main(){
 4      register int num1;
 5      register int num2 = num1;
 6
 7      printf("num1 = %d \n", num1);
 8      printf("num2 = %d \n", num2);
 9
10      return 0;
11  }
```

Listing 4.4: Indeterminate value, test 1

Compiler/tool	Warning/Error	Output

GCC/GCC -O0/1/2/3	warning: 'num1' is uninitialized in this function	num1=0
		num2=0

Clang/Clang -O0	warning: variable 'num1' is uninitialized when used here	num1=0
		num2=0

Clang -O1	warning: variable 'num1' is uninitialized when used here	num1=1858354504
		num2=10678880

Clang -O2	warning: variable 'num1' is uninitialized when used here	num1=-701172936
		num2=18829920

Clang -O3	warning: variable 'num1' is uninitialized when used here	num1=2102302552
		num2=37720672

Intel/Intel -O0/1/2/3	-	num1=0
		num2=0

TCC	-	num1=0
		num2=0

GCC UBSAN/UBSAN -O0/1/2/3	-	num1=0
		num2=0

Clang UBSAN/UBSAN -O0	-	num1=0
		num2=0

Clang UBSAN -O1	-	num1=-1692547352
		num2=0

Clang UBSAN -O2	-	num1=2027954744
		num2=0

Clang UBSAN -O3	-	num1=-350984040
		num2=0

Frama-C	warning: accessing uninitialized left-value	-

Table 4.4: Compilers and tools for test in 4.4

Second test

The complexity of this test is higher and it is harder for compilers and tools to detect the undefined behavior for this example. Objects `num1` and `num2` have indeterminate values (uninitialized). Line 10 has undefined behavior as the program tries to use the value of `num1` which is indeterminate.

```
 1 #include <stdio.h>
 2 
 3 int main(){
 4 
 5     float num1;
 6     float num2;
 7     int a = 1;
 8 
 9     while (a > 0){
10 
11         num2 = 10*num1;  /*a value of an object with automatic storage duration
               is used while it is indeterminate*/
12         printf("num1 = %f \n", num1);
13         printf("num2 = %f \n", num2);
14         a--;
15     }
16 
17     return 0;
18 }
```

Listing 4.5: Indeterminate value, test 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: 'num1' is uninitialized in this function	num1=0.000000 num2=0.000000
Clang/Clang -O0	warning: variable 'num1' is uninitialized when used here	num1=0.000000 num2=0.000000
Clang -O1/2/3	warning: variable 'num1' is uninitialized when used here	num1=0.000000 num2=nan
Intel/Intel -O0/1/2/3	-	num1=0.000000 num2=0.000000
TCC	-	num1=0.000000 num2=0.000000
GCC UBSAN	-	num1=-424560.. num2=-424560..
GCC UBSAN -O0	-	num1=-241137.. num2=-241137..
GCC UBSAN -O1/2/3	-	num1=0.000000 num2=0.000000
Clang UBSAN/UBSAN -O0	-	num1=0.000000 num2=0.000000
Clang UBSAN -O1/2/3	-	num1=0.000000 num2=nan
Frama-C	warning: accessing uninitialized left-value	-

Table 4.5: Compilers and tools for test in 4.5

GCC and Clang in both tests recognize the uninitialized variables but the output of both programs are undefined results. Tiny C and Intel C++ present the same results for both tests and without any warnings. UBSAN for GCC and Clang does not present any run-time errors and the output is undefined as well. Frama-C detects this case.

4.3 Conversion between two pointer types [Number 22]

The conversion between two pointer types produces a result that is incorrectly aligned, invokes undefined behavior. The C99 standard in clause 6.3.2.3 paragraph 7 mention that a pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(){
    uint8_t ip = 9;
    uint8_t *p1 = &ip;
    uint32_t *p2;

    p2 = ( uint32_t * ) p1; /*undefined behavior, possible incompatible
        alignment */
    printf("p2 = %d\n", *p2);

    return 0;
}
```

Listing 4.6: Conversion between two pointer types

Compiler/tool	Warning/Error	Output
GCC	-	p2=-1152506103
GCC -O0	-	p2=760989449
GCC -O1	-	p2=629145609
GCC -O2	-	p2=1771241481
GCC/GCC -O3	-	p2=2132672521
Clang/Clang -O0/1/2/3	-	p2=9
Intel/Intel -O0/1/2/3	-	p2=9
TCC	-	p2=1073987593
GCC UBSAN	runtime error: load of misaligned address 0x7ffc59110ca7 for type 'uint32_t', which requires 4 byte alignement	p2=-1968453879
GCC UBSAN -O0	runtime error: load of misaligned address 0x7ffc59110ca7 for type 'uint32_t', which requires 4 byte alignement	p2=-226912503
GCC UBSAN -O1	runtime error: load of misaligned address 0x7ffc59110ca7 for type 'uint32_t', which requires 4 byte alignement runtime error: load of address 0x7ffc59110ca7 with insufficient space for an object of type 'uint8_t'	p2=896073737
GCC UBSAN -O2	runtime error: load of misaligned address 0x7ffc59110ca7 for type 'uint32_t', which requires 4 byte alignement runtime error: load of address 0x7ffc59110ca7 with insufficient space for an object of type 'uint8_t'	p2=1965031433
GCC UBSAN -O3	runtime error: load of misaligned address 0x7ffc59110ca7 for type 'uint32_t', which requires 4 byte alignement runtime error: load of address 0x7ffc59110ca7 with insufficient space for an object of type 'uint8_t'	p2=-924123127
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: load of address 0x7fff151ae754	p2=9
Frama-C	(line 10) incorrect type for argument 2. The argument will be cast from uint32_t to int (line 10)out of bounds read	-

Table 4.6: Compilers and tools for test in 4.6

For this case, the results differ between compilers. GCC reports different results for any optimization level. Tiny C behaves the same as GCC. Clang and Intel C++ do not report any error or warning and print the seemingly expected result. UBSAN for Clang and GCC detects the undefined behavior during the execution. Frama-C also, reports a message about the case.

4.4 Call function through pointer [Number 23]

If a pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined. In the following example from the university of Carnegie Mellon [Gen18], the C standard library function `strchr()` is called through the function pointer `fp` declared with a prototype with incorrectly typed arguments. According to the C Standard, 6.3.2.3, paragraph 8: "a pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined."

```c
#include <stdio.h>
#include <string.h>

char *(*fp)();

int main() {
  const char *cr;

  fp = strchr;

  cr = fp('e', "Undefined");
  printf("%s\n", cr);
  return 0;
}
```

Listing 4.7: Function call throught pointer

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	Seg fault (core dumped)
Clang/Clang -O0/1/2/3	-	Seg fault (core dumped)
Intel/Intel -O0/1/2/3	-	Seg fault (core dumped)
TCC	-	Seg fault (core dumped)
GCC UBSAN/UBSAN -O0/1/2/3	-	Seg fault (core dumped)
Clang UBSAN/UBSAN -O0/1/2/3	-	Seg fault (core dumped)
Frama-C	pointer to function with incompatible type	-

Table 4.7: Compilers and tools for test in 4.7

This undefined behavior case is impossible for compilers and run-time tools to detect it. Frama-C though reports a message for this case.

4.5 String literal modification [Number 30]

When the program attempts to modify a string literal then the behavior is called undefined.

```
 1  #include <stdio.h>
 2
 3
 4  int main(){
 5      char *p = "undefined behavior number 30";
 6
 7      p[2] = 'L'; /*undefined behavior*/
 8
 9      return 0;
10  }
```

Listing 4.8: Modify string literal

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	Seg fault (core dumped)
Clang/Clang -O0/1/2/3	-	Seg fault (core dumped)
Intel/Intel -O0/1/2/3	-	Seg fault (core dumped)
TCC	-	-
GCC UBSAN/UBSAN -O0/1/2/3	-	Seg fault (core dumped)
Clang UBSAN/UBSAN -O0/1/2/3	7286 ERROR: UndefinedBehaviorSanitizer: SEGV on unkown address	Seg fault (core dumped)
Frama-C	(line 7)out of bound write	-

Table 4.8: Compilers and tools for test in 4.8

In this test example all compilers output, except Tiny C, print 'segmentation fault' without to be able to recognize the undefined behavior. Frama-C and Clang UBSAN recognize the undefined behavior for this program. It is interesting that only Clang UBSAN and not GCC UBSAN recognizes the error.

4.6 Pointer conversion [Number 41]

If a pointer is converted to other than an integer or pointer type, the behavior is undefined. In this test pointer `ptr` points to object `a`. In line 6 `ptr` is converted to double which is undefined behavior according to C99 standard.

```c
#include <stdio.h>

int main(){
    int a = 103;
    int *ptr = &a;
    double dbl = (double) ptr;
    printf("%f\n", dbl);
    return 0;
}
```

Listing 4.9: Pointer conversion

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error(line 6): pointer value used where a floating point value was expected	-
Clang/Clang -O0/1/2/3	error(line 6): pointer cannot be cast to type 'double'	-
Intel/Intel -O0/1/2/3	error(line 6): invalid type conversion: "int *" to "double"	-
TCC	-	125190988.00..
Frama-C	error(line 6): user error: cannot cast from int * to double	-

Table 4.9: Compilers output for test in 4.9

GCC, Intel C++, Clang and Frama-C detect the undefined behavior of pointer conversion. However, Tiny C executes the program and prints a number on the screen.

4.7 Pointer addition/subtraction (first case) [Number 43]

In C99 the behavior is undefined if the addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.

```c
#include <stdio.h>

int main(){
    char ub[18] = "UndefinedBehavior";

    char *ptr1 = ub - 1;     /*undefined behavior*/
    char *ptr4 = ub + 19;    /*undefined behavior*/

    printf("(ub-1) = %s\n", ptr1);
    printf("(ub + 19) = %s\n", ptr4);

    return 0;
}
```

Listing 4.10: Pointer addition/subtraction, case 1

Compiler/tool	Warning/Error	Output
GCC	-	(ub-1)= (ub+19)=??
GCC -O0	-	(ub-1)= (ub+19)=*?
GCC -O1	-	(ub-1)= (ub+19)=e}u
GCC -O2	warning: array subscript -1 is below array bounds of 'char[18]' warning: array subscript 19 is above array bounds of 'char[18]'	(ub-1)= (ub+19)=:?
GCC -O3	warning: array subscript -1 is below array bounds of 'char[18]' warning: array subscript 19 is above array bounds of 'char[18]'	(ub-1)= (ub+19)=$?
Clang	-	(ub-1)= (ub+19)=c?
Clang -O0	-	(ub-1)= (ub+19)=?
Clang -O1	-	(ub-1)= (ub+19)=I?
Clang -O2	-	(ub-1)= (ub+19)=l?
Clang -O3	-	(ub-1)= (ub+19)=??
Intel/Intel -O0/1/3	-	(ub-1)= (ub+19)=
Intel -O2	-	(ub-1)= (ub+19)=q??
TCC	-	(ub-1)= (ub+19)=@
GCC UBSAN/GCC UBSAN -O0/1/2/3	-	similar to GCC
Clang UBSAN/Clang UBSAN -O0/1/2/3	runtime error: index -1 out of bounds for type 'char [18]' runtime error: index 19 out of bounds for type 'char [18]'	similar to Clang
Frama-C	-	-

Table 4.10: Compilers output for test in 4.10

Looking at the table 4.10 we can observe that GCC on optimization levels 2 and 3 is able to detect and show a warning about reading outside of array bounds. Clang, Tiny C and Intel C++ are unable to detect the undefined behavior. UBSAN only for Clang detects the error out of bounds.

4.8 Pointer addition/subtraction (second case) [Number 44]

The C99 mention that the behavior of the program is undefined if the addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated.

```c
#include <stdio.h>

int main(){
    char ub[18] = "UndefinedBehavior";
    char *ptr = ub + 18;    /*pointing to just beyond*/
    char var = *ptr; /*undefined behavior*/
    printf("var = %c\n", var);

    return 0;
}
```

Listing 4.11: Pointer addition/subtraction, case 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O1/2	-	var=?
GCC -O0	-	var=
GCC -O3	-	var=(
Clang/Clang -O0	-	var=?
Clang -O1/2/3	-	var=
Intel/Intel -O2/3	-	var=?
Intel -O0/1	-	var=
TCC	-	var=@
GCC UBSAN/GCC UBSAN -O0/1/2/3	runtime error: load of address 0x7ffd2e4bc442 with insufficient space for an object of type 'char'	similar to GCC
Clang UBSAN/Clang UBSAN -O0	-	similar to Clang
Clang UBSAN -O1/2/3	runtime error: load of address 0x7fffdee8a4c2 with insufficient space for an object of type 'char'	similar to Clang
Frama-C	warning: out of bounds read	-

Table 4.11: Compilers output for test in 4.11

For this case only the tools Frama-C, UBSAN for GCC and for Clang in optimization levels 1-3 recognize the undefined behavior.

4.9 Array subscript [Number 46]

If an array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`), then the behavior is undefined.

First test

In this example a two dimensional array `arr` is declared and initialized. In line 5 the program tries to copy the value of the element in the position `a[0][7]`, but that is out of bounds read and invokes undefined behavior according to C99.

```
1  #include <stdio.h>
2  
3  int main(){
4      int arr[4][5] =
           {{11,12,13,14,15},{21,22,23,24,25},{31,32,33,34,35},{41,42,43,44,45}};
5      int b = arr[0][7]; /*undefined behavior*/
6  
7      printf("%d\n", b);
8      return 0;
9  }
```

Listing 4.12: Array subscript, test 1

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	23
Clang/Clang -O0/1/2/3	warning: array index 7 is past the end of the array (which contains 5 elements)	23
Intel/Intel -O0/1/2/3	-	23
TCC	-	23
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: index 7 out of bounds for type 'int[5]'	23
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: index 7 out of bounds for type 'int[5]'	23
Frama-C	warning: accessing out of bounds index	-

Table 4.12: Compilers output for test in 4.12

In this test, the program copies the element in position `a[0][7]` but this is clearly out of bounds. All compilers and UBSAN print the result 23 which is in the position `a[1][2]`. Clang shows a warning of out of bounds. Also, UBSAN and Frama-C detect the error in this case.

Second test

This example is based on non-compliant code example from Carnegie Mellon university [Seb18a]. This test declares `matrix` to consist of 7 rows and 5 columns in row-major order. The function `init_matrix` iterates over all 35 elements in an attempt to initialize each to the value given by the function argument `x`. However, when the `j` reaches the number of `ROWS` in the first loop(`matrix [0][5]`) will invoke undefined behavior.

```c
#include <stdio.h>
#include <stddef.h>
#define COLS 5
#define ROWS 7

static int matrix[ROWS][COLS];

void init_matrix(int x) {
  for (size_t i = 0; i < COLS; i++) {
    for (size_t j = 0; j < ROWS; j++) {
      matrix[i][j] = x;
    }
  }
}

int main (){
    init_matrix(23);
    return 0;
}
```

Listing 4.13: Array subscript, test 2

Compiler/tool	Warning/Error
GCC/GCC -O0	-
GCC/GCC -O1/2/3	warning: iteration 5 invokes undefined behavior
Clang/Clang -O0/1/2/3	-
Intel/Intel -O0/1/2/3	-
TCC	-
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: index 5 out of bounds for type 'int[5]'
Clang UBSAN/UBSAN -O1/2/3	runtime error: index 5 out of bounds for type 'int[5]'
Frama-C	Warning: accessing out of bounds index

Table 4.13: Compilers output for test in 4.13

This test is too complex for most of the compilers to recognize the undefined behavior. GCC detects only for optimization levels 1-3. UBSAN also detects the error. Frama-C reports a warning about accessing index out of bounds.

4.10 Subtracting two pointers [Number 47]

The behavior is undefined if the result of subtracking two pointers is not representable in an object of type `ptrdiff_t`. In the following test two pointers are used to point-to the first and third element of `arr` respectively. In line 11, a pointer subtraction is happening, that invokes undefined behavior.

```c
 1  #include <stdio.h>
 2
 3  int main(){
 4
 5      int arr[5] = {1, 2 ,3 ,4, 5};
 6
 7      int *ptr1, *ptr2;
 8      ptr1 = &arr[0];
 9      ptr2 = &arr[2];
10
```

```
    ptr1 = ptr2 - ptr1; /*undefined behavior*/

    printf("ptr1 = %d \n", *ptr1);

    return 0;
}
```

Listing 4.14: Subtracting two pointers

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: assignment to 'int *' from 'long int' makes pointer from integer without a cast	Seg fault (core dumped)
Clang/Clang -O0/1/2/3	warning: incompatible integer to pointer conversion assigning to 'int *' from 'long'	Seg fault (core dumped)
Intel/Intel -O0/1/2/3	warning #556: a value of type "long" cannot be assigned to an entity of type "int *"	Seg fault (core dumped)
TCC	warning: assignment makes pointer from integer without a cast	Seg fault (core dumped)
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: load of misaligned address 0x000.. for type 'int', which requires 4 byte alignemnt	Seg fault (core dumped)
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: load of misaligned address 0x000.. for type 'int', which requires 4 byte alignemnt	Seg fault (core dumped)
Frama-C	warning: out of bounds read	-

Table 4.14: Compilers output for test in 4.14

In this case every compiler and tool detect the undefined behavior as a warning and execute the program causing 'segmentation fault'.

4.11 Flexible array member [Number 59]

The C99 language standard mentions that if an attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array, the behavior is undefined. The C99, clause 6.7.2.1, paragraph 16, defines the flexible array member "as a special case, the last element of a structure with more than one named member may have an incomplete array type, this is called flexible array member". In the same paragraph mention that "However, when a . (or ->) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it."[C99]

First test

```
#include <stdio.h>

struct s {
    int n;
    double d[]; /*flexible array member*/
};

int main(){
    struct s t1 = {0};
    t1.d[0] = 4.2;
    printf("%f\n", t1.d[0]);

    return 0;
}
```

Listing 4.15: Flexible array member, test 1

Compiler/tool	Warning/Error	Output
GCC/GCC -O0	-	4.200000 ***stack smashing detected***:<unknown>terminated Aborted (core dumped)
GCC -O1/2/3	-	4.200000
Clang/Clang -O0	-	4.200000
Clang -O1/2/3	-	0.000000
Intel/Intel -O0/1/2/3	-	4.200000
TCC	-	4.200000
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: store to address 0x7ffc.. with insufficient space for an object of type 'double' runtime error: load of address 0x7ffc.. with insufficient space for an object of type 'double'	Same as GCC
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: store to address 0x7ffc.. with insufficient space for an object of type 'double' runtime error: load of address 0x7ffc.. with insufficient space for an object of type 'double'	4.200000
Frama-C	Warning: out of bounds write Failure: TArray with initializer and no length	-

Table 4.15: Compilers output for test in 4.15

It is obvious that the results for this test are vary between compilers. GCC aborts the execution of the program, Clang shows different results for optimization levels 1 to 3 and different results without optimization level. UBSAN tool reports run-time error about insufficient space. Frama-C shows warnings about array out of bounds.

Second test

This test is based on a similar test from the Carnegie Mellon university [Seb18a]. In this test the function `find` try to iterate over the elements of the flexible array member `isbn`, starting from the second element. The program does not allocate any storage for the member `isbn`, the expression `first++` attempts to point the past of the `isbn` when there are no elements.

```c
#include <stdio.h>
#include <stdlib.h>

struct book{
    int value;
    int isbn[]; /*flexible array member*/
};

int *find(struct book *tst, int f){
    int *first = tst->isbn;
    int *last = tst->isbn+0;

    printf("%d\n", *first++); /*undefined behavior*/
    while(first++ != last){
        if(*first == f){
            return first;
        }
    }
    return NULL;
}

int main(){
    int *ptr;
    struct book *tst = (struct book *)malloc(sizeof(struct book));
    if(tst == NULL){
        printf("ERROR\n");
        return 1;
    }
    ptr = find(tst, 10);
    if(ptr != NULL){
        printf("ptr = %d\n", *ptr);
    }
    else{
        printf("returned null\n");
    }
    free(tst);
    return 0;
}
```

Listing 4.16: Flexible array member, test 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	0 Seg fault (core dumped)
Clang/Clang -O0/1	-	0 Seg fault (core dumped)
Clang -O2	-	25485936 ptr=10 returned null
Clang -O1/2/3	-	25485936 ptr=10
Intel/Intel -O0/1/2/3	-	0 Seg fault (core dumped)
TCC	-	0 Seg fault (core dumped)
GCC UBSAN/UBSAN -O0	-	Seg fault (core dumped)
GCC UBSAN -O2/3	runtime error: load of address 0x7ffc.. with insufficient space for an object of type 'int'	Seg fault (core dumped)
Clang UBSAN/UBSAN -O0/1/2/3	UndefinedBehaviorSanitizer:DEADLYSIGNAL SEGV on unknown address. The signal is caused by a READ memory access	0
Frama-C	Warning: out of bounds read	-

Table 4.16: Compilers output for test in 4.16

In this more complex example the results are very different between compilers and optimization levels. Clang optimization level 2 is the most undefined result, it prints all the possible results. Every other compiler output is segmentation fault. UBSAN and Frama-C detect the memory issue about this case.

4.12 Modify const-quialified type [Number 61]

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior of the program is undefined.

```c
1  #include <stdio.h>
2
3
4  int main(){
5      const int val = 10;
6      int *mptr;
7      mptr = &val;
8
9      *mptr = 123; /*modifies constant (was 10). Undefined behavior */
10
11     return 0;
12 }
```

Listing 4.17: Modify const-quialified type

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	warning: assignment discards 'const' qualifier from pointer target type
Clang/Clang -O0/1	warning: assigning to 'int *' from 'const int *' discards qualifiers
Intel/Intel -O0/1/2/3	warning #2332: a value of type "const int *" cannot be assigned to an entity of type "int *"(dropping qualifiers)
TCC	warning: assignment discards qualifiers from pointer target type
GCC UBSAN/UBSAN -O0/1/2/3	-
Clang UBSAN/UBSAN -O0/1/2/3	-
Frama-C	Warning: out of bounds write all target addresses were invalid

Table 4.17: Compilers output for test in 4.17

Every compiler and Frama-C can detect this case of undefined behavior. In line 9 the pointer `mptr` attempts to modify the constant.

4.13 Copying overlapping memory [Number 94]

The behavior is undefined if an attempt is made to copy an object to an overlapping object by use of a library function, other that as explicitly allowed (e.g., `memmove`). The following example attempts to copy 10 bytes, where the destination memory areas overlap by three bytes.

```c
#include <stdio.h>
#include <string.h>

int main(){
    char str[27] = "This is undefined behavior";

    memcpy(str + 7, str, 10);
    printf("After memcpy: %s\n", str);

    return 0;
}
```

Listing 4.18: Copying overlapping memory

Compiler/tool	Warning/Error	Output
GCC/GCC -O0	-	After memcpy:This isThis is hi behavior
GCC -O1/2/3	warning: '__builtin_memcpy' accessing 10 bytes at offsets 7 and 0 overlaps 3 bytes at offset 7	After memcpy:This isThis is hi behavior
Clang/Clang -O0/1/2/3	-	After memcpy:This isThis is hi behavior
Intel/Intel -O0/1/2/3	-	After memcpy:This isThis is hi behavior
TCC	-	After memcpy:This isThis is un behavior
GCC UBSAN/UBSAN -O0/1/2/3	-	After memcpy:This isThis is hi behavior
Clang UBSAN/UBSAN -O0/1/2/3	-	After memcpy:This isThis is hi behavior
Frama-C	Warning: function memcpy: precondition 'separation' got status invalid	-

Table 4.18: Compilers output for test in 4.18

For this undefined behavior case only Frama-C and GCC on optimization levels 1-3 report a warning for overlapping memory.

4.14 Pointer used after free [Number 168]

The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` is used, the behavior is undefined.

```c
#include<stdio.h>
#include<stdlib.h>
#include <string.h>

int main() {
    char str[9] = "tutorial";
    char ftr[9] = "aftertut";
    int bufsize = strlen(str) + 1;
    char *buf = (char *)malloc(bufsize);
    if (!buf) {
        return EXIT_FAILURE;
    }
    free(buf);
    strcpy(buf, ftr); /*undefined behavior*/
    printf("buf = %s\n", buf);

    return EXIT_SUCCESS;
}
```

Listing 4.19: Pointer used after free

Compiler/tool	**Warning/Error**	**Output**
GCC/GCC -O0	-	buf=aftertut
Clang/Clang -O0/1/2/3	-	buf=aftertut
Intel/Intel -O0/1/2/3	-	buf=aftertut
TCC	-	buf=aftertut
GCC UBSAN/UBSAN -O0/1/2/3	-	buf=aftertut
Clang UBSAN/UBSAN -O0/1/2/3	-	buf=aftertut
Frama-C	Warning: accessing left-value that contains escaping addresses	-

Table 4.19: Compilers output for test in 4.19

None of the compilers and UBSAN detect the undefined behavior for this test example. Frama-C reports a warning related to the freed memory.

Chapter 5

Undefined behavior in types

In this section the tests are about undefined behavior that are relevant to types in C. Cases such as declarations errors, demotion of types or incomplete types.

5.1 Object with two declarations [Number 14]

The behavior of a program with two declarations for the same object or function that types are not compatible, is undefined.

First test

In this code example, object `var` at first is declared with type `int`, but in the next line is declared again with type `char` which invokes undefined behavior in this case.

```c
#include <stdio.h>

int main(){
    int var;
    char var; /*undefined behavior*/

    return 0;
}
```

Listing 5.1: Object with two declarations, test 1

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	error: conflicting types for 'var'
Clang/Clang -O0/1/2/3	error: redefinition of 'var' with a different type: 'char' vs 'int'
Intel/Intel -O0/1/2/3	error: "var" has already been declared in the current scope
TCC	error: redeclaration of 'var'
Frama-C	error: redefinition of 'var' in the same scope

Table 5.1: Compilers and tools output for test in 5.1

Second test

For this second more complex test, object `var` declared and initialized in line 9, but in line 13 inside an if statement the object `var` declared again this time with type `long`.

```
1  #include <stdio.h>
2
3  int foo(int a){
4      int b = a^2;
5      return b;
6  }
7
8  int main(){
9      int var = 23;
10     int c = foo(var);
11
12     if(c > 10){
13         long var = 1.0; /*undefined behavior*/
14         printf("%ld\n", var);
15     }
16     printf("%d\n", var);
17
18     return 0;
19 }
```

Listing 5.2: Object with two declarations, test 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	1 23
Clang/Clang -O0/1/2/3	-	1 23
Intel/Intel -O0/1/2/3	-	1 23
TCC	-	1 23
GCC UBSAN/UBSAN -O0/1/2/3	-	1 23
Clang UBSAN/UBSAN -O0/1/2/3	-	1 23
Frama-C	-	-

Table 5.2: Compilers and tools output for test in 5.2

The two tests present different results in the test process. For the first case every compiler and Frama-C recognize the undefined behavior with message "redefinition of the object `var` ". The second test is more complex and the undefined behavior is in line 13, but no compiler or tool detects the case.

5.2 Demotion of a real floating type [Number 16]

The C99 standard under the clause 6.3.1.5 paragraph 2 mention when a `double` is demoted to `float`, a `long double` is demoted to `double` or `float`, if the value being converted is outside the range of values that can be represented, the behavior is undefined.

First test

```
 1  #include <stdio.h>
 2  #include <float.h>
 3
 4  int main(){
 5      double a = 123.134;
 6      double a1 = DBL_MAX;
 7
 8      float b = (float) a; /*undefined behavior*/
 9      float b1 = (float) a1; /*undefined behavior*/
10
11      printf("a= %f\n", a);
12      printf("b= %f\n", b);
13
14      printf("a1= %f\n", a1);
15      printf("b1= %f\n", b1);
16
17      return 0;
18  }
```

Listing 5.3: Demotion of a real floating type, test 1

In this test objects `a` and `a1` with type of `double` demoted to `float` in lines 8 and 9, which invokes undefined behavior.

Compiler/tool	Warning/Error	Output

GCC/GCC -O0/1/2/3	-	a=123.134000
		b=123.134000
		a1=179769..
		b1=inf

Clang/Clang -O0/1/2/3	-	a=123.134000
		b=123.134000
		a1=179769..
		b1=inf

Intel/Intel -O0/1/2/3	-	a=123.134000
		b=123.134000
		a1=179769..
		b1=inf

TCC	-	a=123.134000
		b=123.134000
		a1=179769..
		b1=inf

GCC UBSAN/UBSAN -O0/1/2/3	-	a=123.134000
		b=123.134000
		a1=179769..
		b1=inf

Clang UBSAN/UBSAN -O0/1/2/3	runtime error(line 9): 1.79769e+308 is outside the range of representable values of 'float'	a=123.134000
		b=123.134000
		a1=179769..
		b1=inf

Frama-C	warning: non-finite float value	-

Table 5.3: Compilers and tools output for test in 5.3

Second test

```
1  #include <stdio.h>
2  #include <float.h>
3
4  void func(double da, long double ld) {
5      double db = (float)ld;
6      float fa = (float)da;
7      float fb = (float)ld;
8
9      printf("db= %f\n", db);
10     printf("fa= %f\n", fa);
11     printf("fb= %f\n", fb);
12 }
13
14 int main(){
15     long double alpha = 10.00000;
16     double beta = DBL_MAX;
17     func(beta, alpha);
18
19     return 0;
20 }
```

Listing 5.4: Demotion of a real floating type, test 2

This test example is similar to Carnegie Mellon university example of floating-point conversions [Hed18]. This test attempts to perform conversions (lines 5-7) that may result in truncating values outside the range of the destination types.

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	db=10.000000 fa=inf fb=10.000000
Clang/Clang -O0/1/2/3	-	db=10.000000 fa=inf fb=10.000000
Intel/Intel -O0/1/2/3	-	db=10.000000 fa=inf fb=10.000000
TCC	-	db=10.000000 fa=inf fb=10.000000
GCC UBSAN/UBSAN -O0/1/2/3	-	db=10.000000 fa=inf fb=10.000000
Clang UBSAN/UBSAN -O0/1/2/3	runtime error(line 9): 1.79769e+308 is outside the range of representable values of 'float'	db=10.000000 fa=inf fb=10.000000
Frama-C	warning: non-finite float value	-

Table 5.4: Compilers and tools output for test in 5.4

Both tests present similar results, only UBSAN for Clang and Frama-C detect the undefined behavior for these two tests. Every other compiler and tool executes the program and prints the values on the screen.

5.3 Incomplete type [Number 18]

The behavior is undefined if a non-array lvalue with an incomplete type is used in a context that requires the value of the designated object.

First test

```c
1  #include <stdio.h>
2
3  struct grades{
4      int math;
5      int physics;
6      int literature;
7      struct grades gt; /*incomplete type*/
8  };
9
10 int main(){
11     struct grades grades1; /*incomplete type*/
12     int s = sizeof(grades1);
13     printf("the size of struct grades is: %d\n", s);
14
15     return 0;
16 }
```

Listing 5.5: Incomplete type, test 1

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error(line 7): field 'gt' has incomplete type	-
Clang/Clang -O0/1/2/3	error(line 7): field has incomplete type 'struct grades'	-
Intel/Intel -O0/1/2/3	error(line 7): incomplete type is not allowed	-
TCC	error(line 7): field 'gt' has incomplete type	-
Frama-C	User Error: field ‘gt’ is declared with incomplete type struct grades User Error: type struct grades is circular	-

Table 5.5: Compilers and tools output for test in 5.5

All compilers and Frama-C detect the undefined behavior of incomplete type. We do not use UBSAN for this test because it is a run-time tool and for this test all compilers report an error.

Second test

```c
#include <stdio.h>

int main(){
    int a = 10;
    void b; /*incomplete type according to the standard that cannot be
        completed.*/

    int sa = sizeof(a);
    int sb = sizeof(b); /*undefined behavior*/

    printf("size of a = %d\n", sa);
    printf("size of b = %d\n", sb);

    return 0;
}
```

Listing 5.6: Incomplete type, test 2

In C99 standard clause 6.2.5 (types), paragraph 19 mention that "The void type comprises an empty set of values; it is an incomplete type that cannot be completed."

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error: In function 'main' variable of field 'b' declared void	-
Clang/Clang -O0/1/2/3	error: variable has incomplete type 'void'	-
Intel/Intel -O0/1/2/3	error: incomplete type is not allowed	-
TCC	-	size of a=4 size of b=1
Frama-C	error: using size of 'void'	-

Table 5.6: Compilers and tools output for test in 5.6

For this second test GCC, Clang, Intel C++ recognize the undefined behavior and show an error. Nevertheless, Tiny C executes the program without warnings or errors and prints the `sizeof` values. Frama-C also detects the case. We do not use UBSAN for this test because it is a run-time tool and for this test all compilers report an error.

5.4 Array type with register storage class [Number 19]

If an lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class, the behavior is undefined

```c
#include <stdio.h>

int main(){
    register int arr[4] = {90,91,92,93};
    int *parr = arr;
    printf("parr = %d \n", *parr);

    return 0;
}
```

Listing 5.7: Array type with register storage class

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error(line 5): address of register variable 'arr' requested	-
Clang/Clang -O0/1/2/3	-	parr=90
Intel/Intel -O0/1/2/3	warning #138: taking the address of a register variable is not allowed	parr=90
TCC	-	parr=90
Clang UBSAN/UBSAN -O0/1/2/3	-	parr=90
Frama-C	-	-

Table 5.7: Compilers and tools output for test in 5.7

This test presents interesting results and they differ one from the other. GCC it is obvious that reports an error and detects the undefined behavior. Clang though do not report any warning or error and prints the value of the first array element throught the use of the `parr` pointer. Tiny C and UBSAN for Clang present the same results. Intel C++ prints also the first element of the array but shows a relevant warning. Frama-C do not report anything.

5.5 Use the value of void expression [Number 20]

The behavior is undefined if an attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to void) is applied to a void expression.

```c
#include <stdio.h>

void foo(int alpha, int beta){
    alpha = alpha*beta;
}

int main(){
    int gamma = (int)foo(2, 4); /*undefined behavior, explicit conversion*/
    printf("gamma = %d \n", gamma);

    return 0;
}
```

Listing 5.8: Use the value of void expression

For this test in function `main` the object `gamma` with type `int` is used to call the function `foo` of type `void` but with an explicit conversion to `int` which is undefined behavior according to C99 language standard.

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error(line 8): invalid use of void expression	-
Clang/Clang -O0/1/2/3	error(line 8): operand of type 'void' where arithmetic or pointer type is required	-
Intel/Intel -O0/1/2/3	error(line 8): invalid type conversion: "void" to "int"	-
TCC	-	gamma=8
Frama-C	line 8: Failure: lvalue of type void: tmp	-

Table 5.8: Compilers and tools output for test in 5.8

GCC, Clang, Intel C++ and Frama-C detect the undefined behavior for this test. Tiny C performs the multiplication (2*4) through the call of function `foo` and prints the result 8. We do not use UBSAN for this test because it is a run-time tool and for this test all compilers report an error.

5.6 Constant expression without an integer type [Number 52]

If an expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, or immediately-cast floating constants; or contains casts (outside operands to sizeof operators) other than conversions of arithmetic types to integer types, the behavior is undefined.

```c
#include <stdio.h>

int main(){
    int a = sizeof (struct { int a:((int) (5/3.14f));});
    printf("%d\n", a);

    return 0;
}
```

Listing 5.9: Constant expression does not have an integer type

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: bit-field 'a' width not an integer constant expression	4
Clang/Clang -O0/1/2/3	warning: expression is not an integer constant expression; folding it to a constant is a GNU extension	4
Intel/Intel -O0/1/2/3	-	4
TCC	-	4
GCC UBSAN/UBSAN -O0/1/2/3	-	4
Clang UBSAN/UBSAN -O0/1/2/3	-	4
Frama-C	Failure: bitfield width is not an integer constant	-

Table 5.9: Compilers and tools output for test in 5.9

In this test Frama-C, GCC and Clang detect the undefined behavior of constant expression.

5.7 Function declared at block scope [Number 57]

If a function is declared at block scope with an explicit storage-class specifier other that extern, then the behavior is undefined.

```c
#include <stdio.h>

void foo();

int main(){
    foo();
    fun(1);
    return 0;
}

void foo(){
    void fun(int a); /*undefined behavior*/
}

void fun (int a){
    printf("undefined behavior\n");
}
```

Listing 5.10: Function declared at block scope

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: implicit declaration of function 'fun' warning: conflicting types of 'fun'	undefined behavior
Clang/Clang -O0/1/2/3	warning: implicit declaration of function 'fun' is invalid in C99 error: conflicting types for 'fun' error: conflicting types for 'fun'	-
Intel/Intel -O0/1/2/3	warning#266: function "fun" declared implicitly warning#159: declaration is incompatible with previous "fun"	undefined behavior
TCC	warning: implicit declaration of function 'fun' error: incompatible types for redefinition of 'fun'	-
GCC UBSAN/UBSAN -O0/1/2/3	-	undefined behavior
Frama-C	Warning: Calling function foo that is declared without prototype	-

Table 5.10: Compilers and tools output for test in 5.10

All compilers and tools are able to recognize this undefined behavior, either report an error or a warning.

5.8 No named structure or union member [Number 58]

In C99 standard, clause 6.7.2.1, paragraph 7 mention that "The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. The struct-declaration-list is a sequence of declarations for the members of the structure or union. If the struct-declaration-list contains no named members, the behavior is undefined. The type is incomplete until after the } that terminates the list."

```c
#include <stdio.h>
#include <string.h>

struct Movie {
   char  title[50];
   char  director[50];
   int   rating; /*between 1 and 10*/
};

struct foo {
   int : 12; /*no named structure member*/
};

int main(){
    struct Movie mov1;

    strcpy( mov1.title, "Godfather");
    strcpy( mov1.director, "Kubrick");
    mov1.rating = 10;

    struct foo trial;

    return 0;
}
```

Listing 5.11: No named structure member

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	warning: struct has no named members
Clang/Clang -O0/1/2/3	warning: struct without named members is a GNU extension
Intel/Intel -O0/1/2/3	-
TCC	-
Frama-C	-
GCC UBSAN/UBSAN -O0/1/2/3	-
Clang UBSAN/UBSAN -O0/1/2/3	-

Table 5.11: Compilers and tools output for test in 5.11

Clang and GCC detect that the structure `foo` in 5.11 has no named members. Intel C++, Tiny C, Frama-C and UBSAN do not detect the case in this example.

5.9 Function type includes type qualifiers [Number 63]

In C99 clause 6.7.3 type qualifiers are const, restrict and volatile. In Annex J.2 mention that the behavior is undefined if the specification of a function type includes any type qualifiers.

```c
#include <stdio.h>

const int fun (int a, int b){
    int sum = a + b;
    return sum;
}

int main(){
    int res = fun(23, 34);
    printf("%d\n", res);
    return 0;
}
```

Listing 5.12: No named structure member

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	57
Clang/Clang -O0/1/2/3	-	57
Intel/Intel -O0/1/2/3	warning #858: type qualifier on return type is meaningless	57
TCC	-	57
GCC UBSAN/UBSAN -O0/1/2/3	-	57
Clang UBSAN/UBSAN -O0/1/2/3	-	57
Frama-C	-	-

Table 5.12: Compilers and tools output for test in 5.12

For this case only Intel C++ reports a warning about the type qualifier, every other tool and compiler ignores it and executes the program normally.

5.10 Function with external linkage and inline specifier [Number 67]

If a function with external linkage is declared with an inline function specifier, but is not also defined in the same translation unit, the behavior is undefined. The following code shows two files. The first file includes the file declaration and the second file includes the function definition.

```c
/*File 1*/
#include <stdio.h>

extern inline void fun();

int main(){
    fun();
    return 0;
}
```

Listing 5.13: Inline function declaration

```c
/*File 2*/
#include <stdio.h>

void fun(){
    printf("23\n");
}
```

Listing 5.14: Inline function definition

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: inline fucntion 'fun' declared but never defined	23
Clang/Clang -O0/1/2/3	warning: inline function 'fun' is not defined [-Wundefined-inline]	23
Intel/Intel -O0/1/2/3	-	23
TCC	-	23
GCC UBSAN/UBSAN -O0/1/2/3	-	23
Clang UBSAN/UBSAN -O0/1/2/3	-	23
Frama-C	Warning: Calling function fun that is declared without prototype	-

Table 5.13: Compilers and tools output for test in 5.13 and 5.14

GCC, Clang and Frama-C detect the undefined behavior for this test example, and print a relevant warning.

5.11 Two compatible array types [Number 70]

In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values. The C99 in clause 6.7.5.2, paragraph 6 mention ”If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values”.

```c
#include <stdio.h>

void func(int n){
    int a[10][n];
    int (*p)[20];

    /* Undefined unless n == 20: incompatible types otherwise */

    p = a;
}

int main(){
    func(10);
    return 0;
}
```

Listing 5.15: Two compatible array types

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	-
Clang/Clang -O0/1/2/3	-
Intel/Intel -O0/1/2/3	-
TCC	-
GCC UBSAN/UBSAN -O0/1/2/3	-
Clang UBSAN/UBSAN -O0/1/2/3	-
Frama-C	User Error: Length of array is not a constant: n

Table 5.14: Compilers and tools output for test in 5.15

No compiler and UBSAN detect this example of undefined behavior. However, Frama-C recognize a user error about the length of the array.

5.12 Array with static parameter [Number 71]

If a declaration of an array parameter includes the keyword static within the [and] and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements, then the behavior is undefined. In the following test an array `m_array` is declared with 4 elements and passed as an argument to `foo` function, but this function accepts as argument an array of integers with at least 10 elements.

```
#include <stdio.h>

int foo (int arr[static 10]){
    int s = 0;
    s = s + arr[8];
    return s;
}

int main(){
    int m_arr[4] = {1,1,1,1};
    int y = foo(m_arr); /*undefined behavrior, needs at least 10*/
    printf("y = %d\n", y);

    return 0;
}
```

Listing 5.16: Array with static parameter

Compiler/tool	Warning/Error	Output
GCC	-	y=2141204960
GCC -O0	-	y=-1492422176
GCC/GCC -O1/2/3	-	y=0
Clang/Clang -O0	warning: array argument is too small; contains 4 elements, callee requires at least 10	y=4198848
Clang -O1	warning: array argument is too small; contains 4 elements, callee requires at least 10	y=7
Clang -O2	warning: array argument is too small; contains 4 elements, callee requires at least 10	y=-461991192
Clang -O3	warning: array argument is too small; contains 4 elements, callee requires at least 10	y=-1620823704
Intel	-	y=-829603224
Intel -O0	-	y=4198864
Intel -O1/3	-	y=0
Intel -O2	-	y=-1
TCC	error: identifier expected	-
GCC UBSAN	-	y=997552768
GCC UBSAN -O0	-	y=-1232285056
GCC UBSAN -O2/3	runtime error: load of address 0x7fff.. with insufficient space for an object of type 'int' warning: array subscript 8 is above array bounds of 'int[4]'	y=0
Clang UBSAN/UBSAN -O0	-	y=4338336
Clang UBSAN -O1/2/3	-	y=0
Frama-C	Warning: out of bounds read	-

Table 5.15: Compilers and tools output for test in 5.16

Every compiler show different results for almost every optimization level. Clang prints a warning of small array size. Tiny C compiler presents an error and stops the execution of the program. UBSAN for GCC presents a run-time error for optimization levels 2 and 3. Frama-C reports an out of bounds read warning.

5.13 Scalar initializer [Number 75]

If the initializer for a scalar is neither a single expression nor a single expression enclosed in braces, the behavior is undefined.

```c
#include <stdio.h>

int main(){
    int var = {4,5};
    printf("%d\n", var);

    return 0;
}
```

Listing 5.17: Scalar initializer

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: excess elements in scalar initializer	4
Clang/Clang -O0/1/2/3	warning: excess elements in scalar initializer	4
Intel	warning #1368: excess elements initializer are ignored	4
TCC	error: '}' expected (got, ",")	-
GCC UBSAN/UBSAN -O0/1/2/3	-	4
Clang UBSAN/UBSAN -O0/1/2/3	-	4
Frama-C	User Error: scalar value (of type int) initialized by compound initializer	-

Table 5.16: Compilers and tools output for test in 5.17

All compilers and Frama-C detect the undefined behavior in this test. However, Tiny C reports an error and stops the program execution. UBSAN doesn't detect this case.

5.14 Identifier with two external definitions [Number 78]

The behavior called undefined, if an identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier.

```c
#include <stdio.h>

extern int k;

int main(){
    printf("k = %d\n", k);
    return 0;
}
```

Listing 5.18: Identifier with two external definitions, file 1

```c
#include <stdio.h>
extern int k;
```

Listing 5.19: Identifier with two external definitions, file 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	undefined reference to 'k'	-
Clang/Clang -O0/1/2/3	undefined reference to 'k'	-
Intel	undefined reference to 'k'	-
TCC	error: undefined symbol 'k'	-
Frama-C	-	-

Table 5.17: Compilers and tools output for test in 5.18 and 5.19

Every compiler reports an error for this test. All compilers are able to detect this undefined behavior case. Frama-C though it is not able to detect this case as an erroneous operation.

5.15 Function with identifier list [Number 79]

If a function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list, the behavior is undefined. In the following test example, two functions `func1` and `func2` are defined with identifiers list. Function `func2` accepts three arguments and the identifier list has only two.

```c
#include <stdio.h>

int func1 ( var, nar )
int var;
int *nar;
{
    return var;
}

int func2 ( alpha, beta, gamma ) /*undefined behavior*/
int alpha;
int beta;
{
    return gamma;
}

int main(){

    int *ptr = 0;
    printf("%d\n", func1(10, ptr));

    printf("%d\n", func2(2333, 20));


    return 0;
}
```

Listing 5.20: Function with identifier list

Compiler/tool	Warning/Error	Output
GCC	warning: type of 'gamma' defaults to 'int'	10 213738880
GCC -O0	warning: type of 'gamma' defaults to 'int'	10 -468478592
GCC -O1/2/3	warning: type of 'gamma' defaults to 'int'	10 0
Clang	warning: parameter 'gamma' was not declared, defaulting to type 'int' warning: too few arguments in call to 'func2'	10 -293534336
Clang -O0	warning: parameter 'gamma' was not declared, defaulting to type 'int' warning: too few arguments in call to 'func2'	10 -1919257216
Clang -O1/2/3	warning: parameter 'gamma' was not declared, defaulting to type 'int' warning: too few arguments in call to 'func2'	10 0
Intel	warning #1193: standard requires that parameter "gamma" be given a type by a subsequent declaration warning #165: too few arguments in function call	10 0
TCC	error: too few arguments to function	-
GCC UBSAN/UBSAN -O0/1/2/3	-	Similar to GCC
Clang UBSAN/UBSAN -O0/1/2/3	-	Similar to Clang
Frama-C	User Error: Too few arguments in call to func2	-

Table 5.18: Compilers and tools output for test in 5.20

All compilers and Frama-C detect the undefined behavior. However, all compilers execute the program with some warnings except Tiny C that stops the execution with an error.

5.16 Identifier with incomplete type [Number 83]

If an identifier for an object with internal linkage and an incomplete type is declared with a tentative definition, the behavior is undefined. In C99 standard clause 6.9.2 ”External object definitions”, paragraph 2 mention that a declaration for an object has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier static, constitutes a tentative definition.

```c
#include <stdio.h>

int main(){
    static char arr[]; /*incomplete type and tentative definition, internal
       linkage*/

    return 0;
}
```

Listing 5.21: Identifier with incomplete type

Compiler/tool	Warning/Error
GCC/GCC -O0/1/2/3	error: array size missing in ’arr’
Clang/Clang -O1/2/3	error: definition of variable with array type needs an explicit size or an initializer
Intel	error: incomplete type is not allowed
TCC	error: unknown type size
Frama-C	-

Table 5.19: Compilers and tools output for test in 5.21

All compilers detect the error of this test example, reporting several errors. Frama-C does not report anything related to this undefined behavior case.

5.17 Assert macro [Number 105]

If the argument to the assert macro does not have a scalar type, the behavior is undefined.

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>

struct Person {
   char  fname[10];
   char  lname[10];
};

int main () {

    struct Person myPer;
    strcpy( myPer.fname, "Vasilis");
    strcpy( myPer.lname, "Gemistos");

    assert(myPer);
    printf("not stopped\n");

    return(0);
}
```

Listing 5.22: Assert macro

Compiler/tool	**Warning/Error**	**Output**
GCC/GCC -O0/1/2/3	error: used struct type value where scalar is required	-
Clang/Clang -O0/1/2/3	error: used type 'struct Person' where arithmentic or pointer type is required	-
Intel/Intel -O0/1/2/3	error: expression must have arithmetic or pointer type	-
TCC	error: invalid aggregate type for register load	-
Frama-C	Failure: myPer != 0	-

Table 5.20: Compilers and tools output for test in 5.22

All compilers and Frama-C detect the undefined behavior for this test example. We do not use UBSAN for GCC and Clang for this test because GCC and Clang report an error for all optimization levels.

Chapter 6

Undefined behavior in syntax

In this section all undefined behavior cases involve syntax errors, misformed types or keywords. Tests such as header name without delimeters or use of a reserved keyword, will be encountered.

6.1 Unmatched ' or " [Number 24]

The behavior is undefined if an unmatched ' or " character is encountered on a logical source line during tokenization.

```c
#include <stdio.h>

int foo(int a, int b){
    return (a+b);
}

int main(){
    int res = foo(10, 10);

    if(res > 1){
        char c = 'b;
    }

    return 0;
}
```

Listing 6.1: Unmatched ' or "

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: missing terminating ' character error: missing terminating ' character	-
Clang/Clang -O0/1/2/3	warning: missing terminating ' character error: expected expression error: expected ';' at end of declaration	-
Intel/Intel -O0/1/2/3	error: missing closing quote error: expected a ";"	-
TCC	error: missing terminating ' character	-
Frama-C	warning: missing terminating ' character Can't process annotation: unterminated char	-

Table 6.1: Compilers and tools output for test in 6.1

Every compiler and tool detects the unmatched character in this case.

6.2 Reserved Keyword [Number 25]

A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword, the behavior of the program is undefined. In the following example 6.2 we use the reserved keyword `switch` as a function name, which is undefined behavior according to C standard.

```c
#include <stdio.h>

void switch(int a, int b){
    printf("Undefined\n");
}

int main(){
    switch(10, 20);
    return 0;
}
```

Listing 6.2: Reserved keyword

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error(line 3): expected identifier or '(' before 'switch' warning(line 8): left-hand operand of comma expression has no effect	-
Clang/Clang -O0/1/2/3	error(line 3): expected identifier or '('	-
Intel/Intel -O0/1/2/3	error(line 3): expected an identifier	-
TCC	-	-
Frama-C	syntax error: line 3, between columns 0 and 5, before or at token: switch	-

Table 6.2: Compilers and tools output for test in 6.2

Every compiler and tool detects the syntax error of the use of reserved keyword, except Tiny C, that executes the program without errors.

6.3 Character designating a digit [Number 27]

According to C99 standard the initial character of an identifier is a universal character name designating a digit, is undefined behavior.

```c
#include <stdio.h>

int main(){
    int \u0660abc = 10; /*undefined behavior*/
    printf("%d \n", \u0660abc);

    return 0;
}
```

Listing 6.3: Character designating a digit

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error: universal character \u0660 is not valid at the start of an identifier	-
Clang/Clang -O0/1/2/3	error(line 4): expected identifier or '('	-
Intel/Intel -O0/1/2/3	warning #1179: this universal character cannot begin an identifier	10
TCC	error: stray '\' in program	-
Frama-C	syntax error: line 3, between columns 0 and 5, before or at token: switch	-

Table 6.3: Compilers and tools output for test in 6.3

Only Intel C++ executes the program with one warning and prints the result on the screen. The other compilers and tools detect the error.

6.4 Non-significant characters [Number 28]

The behavior is undefined when two identifiers differ only in nonsignificant characters. According to the C99 standard in 5.2.4.1 ”Translation limits” mention that:

- 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)
- 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)

For this case each compiler can set its own significant bytes rules.

```c
#include <stdio.h>

int main(){
    int global_symbol_definition_lookup_table_amsterdam_university_master_A =
        10;
    int global_symbol_definition_lookup_table_amsterdam_university_master_B =
        11;

    printf("%d \n",
        global_symbol_definition_lookup_table_amsterdam_university_master_A);
    printf("%d \n",
        global_symbol_definition_lookup_table_amsterdam_university_master_B);

    return 0;
}
```

Listing 6.4: Non-significant characters

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	10 11
Clang/Clang -O0/1/2/3	-	10 11
Intel/Intel -O0/1/2/3	-	10 11
TCC	-	10 11
Frama-C	-	-

Table 6.4: Compilers and tools output for test in 6.4

No compiler recognize the error of non-significant characters due to reason that is implementation based problem. Compilers may use the a larger number of non-significant bytes and so it is not possible to create a fully correct test for this case.

6.5 Identifier __func__ declared explicitly [Number 29]

The identifier `__func__` is explicitly declared then the behavior of the program is undefined according to the C99 standard. In C99 clause 6.4.2.2 ”Predefined identifiers”, in footnote 62 mention that ”since the name `__func__` is reserved for any use by the implementation, if any other identifier is explicitly declared using the name `__func__`, the behavior is undefined".

```c
#include <stdio.h>

void __func__() { /*undefined behavior*/
    int a = 10;
}

int main() {
    __func__();
    return 0;
}
```

Listing 6.5: Identifier `__func__` declared explicitly

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error: expected identifier or '(' before '`__func__`' error: called object '`__func__`' is not a function or function pointer	-
Clang/Clang -O0/1/2/3	error: expected identifier or '(' error: called object type 'const char [5]' is not a function or function pointer	-
Intel/Intel -O0/1/2/3	error: expected an identifier error: expression preceding parentheses of apparent call must have (pointer-to-) function type	-
TCC	error: function pointer expected	-
Frama-C	syntax error. Location line 8, before or at token `__func__`	-

Table 6.5: Compilers and tools output for test in 6.5

For this test all compilers detect the syntax error of the identifier `__func__`.

6.6 Header name [Number 31]

If the characters ',\, ", //, or /* occur in the sequence between the <and >delimiters, or the characters ',\, ", //, or /* occur in the sequence between the " delimiters, in a header name preprocessing token, the behavior is undefined.

```c
#include <stdio.h>
#include "f\oo.h"

int main(){
    printf("%d \n", a);
    return 0;
}
```

Listing 6.6: Header name

```c
#ifndef f\oo
#define f\oo
    int a = 34;
#endif
```

Listing 6.7: Header file

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: extra tokens at end of #ifndef directive warning: ISO C99 requires whitespace after the macro name	34
Clang/Clang -O0/1/2/3	warning: extra tokens at end of #ifndef directive warning: ISO C99 requires whitespace after the macro name	34
Intel/Intel -O0/1/2/3	warning #14: extra text after expected end of preprocessing directive warning #2217: white space is required between the macro name "f" and its replacement text	34
TCC	error: stray '\' in program	-
GCC UBSAN/UBSAN -O0/1/2/3	-	34
Clang UBSAN/UBSAN -O0/1/2/3	-	34
Frama-C	warning: extra tokens at end of #ifndef directive warning: ISO C99 requires whitespace after the macro name	-

Table 6.6: Compilers and tools output for test in 6.6

For this test GCC, Clang, Intel C++ and Frama-C report the relevant warnings and execute the program. Tiny C reports an error and stops the execution. UBSAN doesn't report any message for this case.

6.7 The closing curly bracket that terminates the function is reached [Number 82]

If the curly bracket that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined. In the following test example, the `func` function checks if a given integer is an element in the array `arr`. If the `func` doesn't find the element in the array the function terminates without returning anything, which invokes undefined behavior.

```c
#include <stdio.h>

int func(int *ptr, int size, int giv){
    for (int i=0; i < size; i++){
        if(ptr[i] == giv){
            return 1;
        }
    }
}

int main(){
    int arr[3] = {1,24,810};
    int c = 23;
    int val = func(arr, sizeof(arr), c);
    printf("%d\n", val);

    return 0;
}
```

Listing 6.8: Function } is reached

Compiler/tool	Warning/Error	Output
GCC/GCC -O0	warning: control reaches end of non-void function	12
GCC -O1	warning: control reaches end of non-void function	-2000440692
GCC -O2/3	warning: control reaches end of non-void function	1
Clang/Clang -O0	warning: control may reach end of non-void function	0
Clang -O1/2/3	warning: control may reach end of non-void function	1
Intel/Intel -O0/1/2/3	warning #1011: missing return statement at end of non-void function "func"	1
TCC	-	12
GCC UBSAN/UBSAN -O0	-	12
GCC UBSAN -O1	-	0
GCC UBSAN -O2/3	run-time error: load of address 0x7ffcd.. with insufficient space for an object of type 'int'	1
Clang UBSAN/UBSAN -O0	-	0
Clang UBSAN -O2/3	-	1
Frama-C	Warning: out of bounds read	-

Table 6.7: Compilers and tools output for test in 6.8

All compilers except Tiny C, report a warning about the non-void function. The printed results are different between compilers and optimization levels. GCC UBSAN reports a run-time error for optimization levels 2 and 3. Also, Frama-C reports a warning about out of bounds read.

6.8 Header name forms [Number 85]

The `#include` preprocessing directive that results after expansion does not match one of the two header name forms. In C99 standard clause 6.10.2, paragraph 7 mention the most common uses of `#include` preprocessing directives are:

- `#include <stdio.h>`
- `#include "myprog.h"`

```c
#include <stdio.h>
#include string.h /*undefined behavior*/

int main(){
    printf("Hello World\n");
    return 0;
}
```

Listing 6.9: Header name forms

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	error: #include expects "FILENAME" or <FILENAME>	-
Clang/Clang -O0/1/2/3	error: expected "FILENAME" or <FILENAME>	-
Intel/Intel -O0/1/2/3	catastrophic error: expected a file name	-
TCC	error: '#include' expects "FILENAME" or <FILENAME>	-
Frama-C	error: #include expects "FILENAME" or <FILENAME>	-

Table 6.8: Compilers and tools output for test in 6.9

For this test example all compilers report an error of misformed header name.

6.9 Library function with invalid value or type [Number 102]

An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments, the behavior is undefined.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main () {
   int val;
   char str[20];

   strcpy(str, "10348593");
   val = atoi(str);
   printf("String=%s, Int=%d\n", str, val);

   strcpy(str, "something else");
   val = atoi(str);
   printf("String= %s, Int=%d\n", str, val);

    /*undefined section*/
    char *ptr = NULL;
    val = atoi(ptr);
    return(0);
}
```

Listing 6.10: Library function with invalid value or type

Compiler/tool	Warning/Error	Output

GCC/GCC -O0/1/2/3	-	String=10348593,Int=10348593
		String=something else,Int=0
		Seg fault(core dumped)

Clang/Clang -O0/1/2/3	-	String=10348593,Int=10348593
		String=something else,Int=0

Intel/Intel -O0/1/2/3	-	String=10348593,Int=10348593
		String=something else,Int=0
		Seg fault(core dumped)

TCC	-	String=10348593,Int=10348593
		String=something else,Int=0
		Seg fault(core dumped)

GCC UBSAN/UBSAN -O0/1/2/3	run-time error: null pointer passed as argument 1, which is declared to never be null	String=10348593,Int=10348593
		String=something else,Int=0
		Seg fault(core dumped)

Clang UBSAN/UBSAN -O0/1/2/3	run-time error: null pointer passed as argument 1, which is declared to never be null	String=10348593,Int=10348593
		String=something else,Int=0

Frama-C	Warning: function atoi: precondition 'valid_nptr' got status invalid	-

Table 6.9: Compilers and tools output for test in 6.10

For this test example only UBSAN and Frama-C detect the undefined behavior. Compilers output differ between Clang and GCC, Tiny C and Intel C++. Clang is the only compiler that doesn't show segmentation fault for the undefined section of 6.10.

Chapter 7

Undefined behavior in arithmetic operations

In this section the cases are related to arithmetic operations.

7.1 Signed integer overflow [Number 33]

If an exception condition occurs during the evaluation of an expression the behavior is undefined. According to C99 standard in the clause 6.5 ”Expressions”, paragraph 5 mention that ”If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined”.

First test

```
#include <stdio.h>
#include <limits.h>

void fun(int a){
    int b = INT_MAX + a;
    printf("b+1 = %d\n", b);
    return;
}

int main(){
    fun(20);
    return 0;
}
```

Listing 7.1: Signed integer overflow, test 1

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	b+1=-2147483629
Clang/Clang -O0/1/2/3	-	b+1=-2147483629
Intel/Intel -O0/1/2/3	-	b+1=-2147483629
TCC	-	b+1=-2147483629
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: signed integer overflow 20 + 2147483647 cannot be represented in type ’int’	b+1=-2147483629
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: signed integer overflow 2147483647 + 20 cannot be represented in type ’int’	b+1=-2147483629
Frama-C	warning: signed integer overflow	-

Table 7.1: Compilers and tools for test in 7.1

In this test only UBSAN and Frama-C detect the error of signed integer overflow. Compilers report as a result a negative number because the result of the operation of adding one number to the biggest integer, cannot be represented.

Second test

The following test example is similar to Regehr's example [Reg10b]

```c
#include <stdio.h>
#include <limits.h>

int fun(int x){
    if(x<0) x = -x;
        return x >= 0;
    }

int main(){
    printf("%d\n", -INT_MIN);
    printf("%d\n", fun(INT_MIN));


    return 0;
}
```

Listing 7.2: Signed integer overflow, test 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: integer overflow in expression '-2147483648' of type 'int' results in '-2147483648'	-2147483648 0
Clang/Clang -O0	-	-2147483648 0
Clang/Clang -O1/2/3	-	-2147483648 1
Intel/Intel -O0/1/2/3	warning #61: integer operation result is out of range	-2147483648 0
TCC	-	-2147483648 0
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: negation of -2147483648 cannot be represented in type 'int'; cast to an unsigned type to negate this value to itself	-2147483648 0
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: negation of -2147483648 cannot be represented in type 'int'; cast to an unsigned type to negate this value to itself	-2147483648 0
Frama-C	warning: signed integer overflow	-

Table 7.2: Compilers and tools for test in 7.2

In this test GCC, Intel C++, UBSAN and Frama-C detect the undefined behavior case.

7.2 Division by zero [Number 42]

The behavior is undefined if the value of the second operand of the / or % operator is zero. The C99 in clause 6.5.5 "Multiplicative operators", in paragraph 5 mention that "The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined".

First test

```c
 1  #include <stdio.h>
 2
 3  int main (){
 4      int n = 20;
 5      int b = 1;
 6
 7      for(int i = 5; i >= 0; i--){
 8          n = n * i;
 9          b = n/n;
10          if(n == 0)
11              printf("n = %d, b(n/n) = %d\n", n, b);
12      }
13      return 0;
14  }
```

Listing 7.3: Division by zero, test 1

In this example, inside the `for` loop the operation `n/n (for n==0)` will cause undefined behavior. The program asks to print the value of objects `n` and `b` for the case that `n==0` in line 9.

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	n=0, b(n/n)=1
Clang/Clang -O0	-	Floating point exception (core dumped)
Clang -O1/2/3	-	n=0, b(n/n)=1
Intel/Intel -O0/1/2/3	-	Floating point exception (core dumped)
TCC	-	Floating point exception (core dumped)
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: division by zero	n=0, b(n/n)=1
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: division by zero	n=0, b(n/n)=1
Frama-C	Warning: division by zero	-

Table 7.3: Compilers and tools for test in 7.3

We can observe that Frama-C and UBSAN are able to recognize the undefined behavior of division by zero.

Second test

For this test, the program asks the user for input in order to be more complex for compilers and tools to detect the undefined behavior.

```c
#include <stdio.h>

int func(){
    int gv;
    printf("Enter a integer number: ");
    scanf("%d", &gv);
    return gv;
}

int main (){
    int n = (23/func());
    printf("%d\n", n);

    return 0;
}
```

Listing 7.4: Division by zero, test 2

Compiler/tool	Warning/Error	Output
GCC/GCC -O0	-	Floating point exception (core dumped)
GCC -O1/2/3	warning: ignoring return value of 'scanf', declared with attribute warn_unused_result	Floating point exception (core dumped)
Clang/Clang -O0/1/2/3	-	Floating point exception (core dumped)
Intel/Intel -O0/1/2/3	-	Floating point exception (core dumped)
TCC	-	Floating point exception (core dumped)
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: division by zero	Floating point exception (core dumped)
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: division by zero	Floating point exception (core dumped)
Frama-C	Warning: division by zero	-

Table 7.4: Compilers and tools for test in 7.4

The results of this test are similar to 7.3, Frama-C and UBSAN detect the undefined behavior case. GCC for optimization levels 1 - 3 shows a non-relevant to the case warning.

7.3 Bit shifting [Number 48]

If an expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression, the behavior is undefined.

```c
#include <stdio.h>

int main(){
    int x = 5 << -3;
    int y = 5 >> -3;

    printf("x= %d , y = %d\n", x, y);

    return 0;
}
```

Listing 7.5: Bit shifting

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	warning: left shift count is negative warning: right shift count is negative	x=0,y=40
Clang/Clang -O0	warning: shift count is negative warning: shift count is negative	x=4198704,y=4198704
Clang -O1	warning: shift count is negative warning: shift count is negative	x=1220399448,y=1220399464
Clang -O2	warning: shift count is negative warning: shift count is negative	x=-928258344,y=-928258328
Clang -O3	warning: shift count is negative warning: shift count is negative	x=-1591986904,y=-1591986888
Intel/Intel -O0/1/2/3	warning: shift count is negative warning: shift count is negative	x=-1610612736,y=0
TCC	-	x=-1610612736,y=0
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: shift exponent -3 is negative	Same as GCC
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: shift exponent -3	Similar to Clang
Frama-C	Warning: invalid RHS operand for shift	-

Table 7.5: Compilers and tools for test in 7.5

Every compiler and tool detects the undefined behavior, except Tiny C.

7.4 Size expression in an array declaration [Number 69]

If the size expression in an array declaration is not a constant expression and evaluates at program execution time to a non-positive value, the behavior is undefined.

```c
#include <stdio.h>

int func(int n){
    int arr[n]; /*undefined behavior*/
    return (sizeof(arr));
}

int main(){
    printf("func(0) = %d\n", func(0));
    printf("func(-10) = %d\n", func(-10));

    return 0;
}
```

Listing 7.6: Size expression in an array declaration

Compiler/tool	Warning/Error	Output
GCC/GCC -O0/1/2/3	-	func(0)=0 func(-10)=-40
Clang/Clang -O0/1/2/3	-	func(0)=0 func(-10)=-40
Intel/Intel -O0/1/2/3	-	func(0)=0 func(-10)=-40
TCC	-	func(0)=0 func(-10)=-40
GCC UBSAN/UBSAN -O0/1/2/3	runtime error: variable length array bound evaluates to non-positive value 0	func(0)=0 func(-10)=-40
Clang UBSAN/UBSAN -O0/1/2/3	runtime error: variable length array bound evaluates to non-positive value 0	func(0)=0 func(-10)=-40
Frama-C	Warning(line 4): assertion 'alloca_bounds' got status invalid	-

Table 7.6: Compilers and tools for test in 7.6

UBSAN and Frama-C detect the undefined behavior for the test in 7.6.

Chapter 8

Discussion

In this chapter we discuss the findings from the tests results. We present how many tests each compiler and each tool detects. Many undefined behavior cases are really important to investigate and programmers could receive help from this study in order to know how to avoid certain errors and issues. It could benefit developers to know in depth the behavior of their programs and how to navigate around undefined behavior cases. Many undefined behavior cases are dangerous and can lead to security issues, as mentioned in Chapter 1.

A program is meaningless with undefined behavior. Many open source projects and industrial projects fail systematically due to undefined behavior. This thesis presents a catalogue of 50 undefined behavior cases with tests for every one of them. These tests could be a guide to programmers to understand in depth the possible problems that a code with undefined behavior can cause by looking at compilers and tools output in our study. From the tests in Chapters 3 - 7 we can observe that in many cases the tested compilers and tools present undefined results, even in simple cases. That means every undefined behavior, even the simple ones, can harm an entire project. Undefined results are very hard to be interpreted by programmers. This can be time-consuming until programmers are able to detect the undefined behavior during the debug process. However, there are cases mentioned in Chapter 6 that there is no need to be considered as undefined behavior. Cases with syntax errors are very easy to be detected by compilers and tools. The language committee should consider to mark those cases as errors instead of undefined behavior. We present one example of a case like this, an unmatched ' or " is said to have undefined behavior as mentioned in 6.1. In this example every compiler and tool in our study detects this case as a syntax error. Cases similar to this example could be marked as errors.

In this thesis we present 50 undefined behavior cases and overall 62 tests. In the following table, we record the number of detected undefined behavior tests per compiler and tool.

Compiler/tool	**Detectable cases**
GCC	38/62
Clang	36/62
Intel	28/62
TCC	21/62
Frama-C	49/62

Table 8.1: Detected undefined behavior cases out of 62 tests per compiler and tool

In table 8.1 we can observe that GCC and Clang present similar numbers of detected tests. GCC is slightly more efficient and recognizes 38 out of 62, that means 61.3% and Clang detects 36 out of 62 and that means 58%. Intel C++ detects 28 out of 62, that means 45.1%, Tiny C detects 21 out of 62, that means 33.8 %. We see that UBSAN for GCC detects 18 out of tested cases and UBSAN for Clang 19. Finally, Frama-C reports the best result with 49 out of 62 cases, a percentage of 79%. Overall, the most efficient tool in our study is Frama-C and it detects the majority of the undefined behavior tests. UBSAN presents the same results for both compilers GCC and Clang. The most efficient compiler in our study is GCC with a small difference from Clang. Intel C++ and Tiny C detect the fewest cases, with 28 and 21 detected tests respectively.

8.1 Differences between C99 and C11

In this section we report the new undefined behavior cases or the cases that have been modified from C99 to C11 standard [C11]. At first, we present the new entries in undefined behavior section of C11.

- The execution of a program contains a data race (5.1.2.4)
- A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7)
- An lvalue designating an object of automatic storage duration that could have been declared with the register storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1)
- A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5)
- A member of an atomic structure or union is accessed (6.5.2.3)
- A function declared with a _Noreturn function specifier returns to its caller (6.7.4)
- The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5)
- Declarations of an object in different translation units have different alignment specifiers (6.7.5)
- A signal handler called in response to SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1)
- The signal function is used in a multi-threaded program (7.14.1.1)
- The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than INT_MAX (7.21.6.1, 7.29.2.1)
- The alignment requested of the aligned_alloc function is not valid or not supported by the implementation, or the size requested is not an integral multiple of the alignment (7.22.3.1)
- A signal is raised while the quick_exit function is executing (7.22.4.7)
- At least one member of the broken-down time passed to asctime contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.27.3.1)
- When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded. 105) If the quotient a/b is representable, the expression (a/b)*b + a%b shall equal a; otherwise, the behavior of both a/b and a%b is undefined.

The latter case is not displayed in the undefined behavior J annex in C11 but is mentioned in section 6.5.5, clause 6. Next, we present the cases that have been modified along with the number of C99 list (can be found in A):

- An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, _Alignof expressions, or immediately-cast floating constants; or contains casts (outside operands to sizeof and _Alignof operators) other than conversions of arithmetic types to integer types (6.6) [Number 52]
- An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, or _Alignof expressions; or contains casts (outside operands to sizeof or _Alignof operators) other than conversions of arithmetic types to arithmetic types (6.6). [Number 54]
- A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1) [Number 58]
- A signal occurs other than as the result of calling the abort or raise function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as volatile sig_atomic_t, or calls any function in the standard library other than the abort function, the _Exit function, the quick_exit function, or the signal function (for the same signal number) (7.14.1.1) [Number 125]
- The program calls the exit or quick_exit function more than once, or calls both functions (7.22.4.4, 7.22.4.7) [Number 172]
- During the call to a function registered with the atexit or at_quick_exit function, a call is made to the longjmp function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7)

[Number 173]

Finally there are two undefined behavior cases that have been removed in the C11 standard:

- Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored (6.5) [Number 32]
- An attempt is made to modify the result of a function call, a conditional operator, an assignment operator, or a comma operator, or to access it after the next sequence point (6.5.2.2, 6.5.15, 6.5.16, 6.5.17) [Number 35]

Chapter 9

Related work

The Software Engineering Institute of the Carnegie Mellon University created an online list of undefined behavior from the C11 international language standard. In many undefined behavior cases they addressed coding practices with compliant and non compliant examples. Also, listed cases have been marked as critical, bounded, undefined behavior (information/confirmation needed) and possible conforming language extension [Seb18b].

Wang et al. proposed a static checker tool called STACK that identifies undefined behavior bugs in programming languages such as C/C++. They applied their tool to widely used systems and the STACK identified 161 new bugs. They tested a subset of undefined behavior from C11 standard, which they called it C*. This article presents the first systematic study of unstable code [WZKSL15]. Wang in another study presented the reasons and vulnerabilities [WCC+12]. C compilers trust the programmer not to write code with undefined behavior and they optimize the code under this assumption. Authors claim that experienced C programmers know well that code with undefined behavior can result in surprising findings, and many compilers support flags to selectively disable certain optimizations that exploit undefined behavior.

Regehr is actively researching undefined behavior in C and C++. Regehr et al. presented a compilers testing tool, Csmith that tests a large subset of C programming language while avoiding undefined and unspecified behaviors. Authors try to exclude all of 191 kinds of undefined behavior and 52 kinds of unspecified behavior that are listed in the C99 standard [YCER11]. We don't use Csmith and STACK tools because of our limited time to complete this thesis.

Undefined behavior facilitates optimizations by permitting a compiler to assume that programs will only execute defined operations, and they also support error checking since a conforming compiler implementation can cause the program to abort with a diagnostic when an undefined operation is executed. Y. Song et al. present a study for the role of undefined behavior in known compilers such as GCC, LLVM, Intel's and Microsoft's which they support one or more forms of undefined behavior. The study focuses on creating solutions for problem of LLVM's intermediate representation with undefined behavior [LKS+17].

Chapter 10

Conclusions and future work

In this thesis we investigate how undefined behavior in C can be detected and classified. We use GCC, Clang, Intel C++, Tiny C compilers and run-time tool UBSAN for Clang and GCC and static analysis tool Frama-C. For this research we select the C99 language standard in order to align with the MISRA catalogue that Solid Sands uses. This catalogue documents the cases from the C99 standard that are detectable by the MISRA tool. C99 has listed 191 undefined behavior, this list can be found in Appendix A. We select 50 undefined behavior cases and we create 62 tests to observe the behavior of the selected compilers and tools. For every test in this study, we report the results of the tested compilers and tools.

In this thesis we found that GCC is the most efficient compiler in detecting undefined behavior in C language programs. We run 62 tests and GCC detects 61.3% of them. Clang detects fewer cases than GCC, with a percent of 58% detected cases. Intel C++ with 45.1% and Tiny C with 33.8% are the most lenient compilers in undefined behavior detection. We found that it is possible an undefined behavior case can be detected by a compiler on a certain optimization level, while the same compiler on a different optimization level, the same case can be undetected. A case like this is mentioned in 4.7. Also, we present cases that no compiler or tool we use can detect. Overall, this thesis could be a guide to programmers to learn how to avoid undefined behavior in C under the tested compilers and tools. In addition, this thesis could contribute to interested professionals who want to create a tool or a compiler for undefined behavior detection. They could use this thesis as a guideline of which undefined behavior cases are detectable. Furthermore, they can check if their compiler or tool presents similar behavior as the existing compilers and tools we use for this thesis.

To finalize our study, we are going to answer the research questions that mentioned in 1.5:

How can we classify if an undefined behavior is detectable statically, at run-time or not detectable? At first, we can create tests about undefined behavior cases and select the desired compilers and tools that we will use to run our tests. For this project after the tests results, we present which compiler or tool detected each undefined behavior case. Many times in this study compilers warn us about a possible issue that is relevant to the undefined behavior. However, there are cases that no compiler and tool was able to detect the error in the test. Those cases are said to be undetectable or the tools that we use in our study are not powerful enough to catch those cases.

What conclusions can we make/extract from the tests of undefined behavior about compilers and tools? In this thesis, many times compilers and tools behave differently for the same test, less often the same compiler can present different behavior on different optimization level. The meaning of this, is that undefined behavior are really hard to predict how the compiler will behave and programmers must be aware about the dangers and issues that can occur from a code with undefined behavior.

10.1 Future work

In this section we discuss how this research can evolve in the future and what aspects can be better developed over time. In our study we developed tests based on undefined behavior and tested four compilers and two tools. Interesting results can arise if more compilers and more versions of the same compiler can be tested. For example GCC and Clang offer a lot of versions. Through this direction, we can observe how compilers evolve and train to catch and detect undefined behavior over time. In addition, more undefined behavior cases can be tested from the C99 standard. The ultimate goal will be to create a full catalogue of all undefined behavior cases from C99 or the newest standard. However, a nice implementation would be compliant and non-compliant code examples as we saw in Carnegie Mellon university examples [Seb18b]. Nevertheless, optimizations and undefined behavior is an important and interesting aspect. Compilers can optimize a program with undefined behavior and discard code containing undefined behavior [Lat11b].

Bibliography

[C11] ISO/IEC 9899:201x, International Standard, Programming Languages – C. https://webstore.ansi.org/Standards/ISO/ISOIEC98992011.

[C99] ISO/IEC 9899:TC3, International Standard, Programming Languages – C.

[CER] C compilers may silently discard some wraparound checks. https://www.kb.cert.org/vuls/id/162289/.

[Cla] Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

[FRA] Frama-C. https://frama-c.com/.

[GCC] GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[Gen18] Jeff Gennari. EXP37-C. Call functions with the correct number and type of arguments. https://wiki.sei.cmu.edu/confluence/display/c/EXP37-C.+Call+functions+with+the+correct+number+and+type+of+arguments, December 2018.

[Hed18] Shaun Hedrick. FLP34-C. Ensure that floating-point conversions are within range of the new type. https://wiki.sei.cmu.edu/confluence/display/c/FLP34-C.+Ensure+that+floating-point+conversions+are+within+range+of+the+new+type, December 2018.

[ICC] Intel C++ Compiler. https://software.intel.com/en-us/c-compilers.

[Lat11a] Chris Lattner. What Every C Programmer Should Know About Undefined Behavior 1/3. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html, May 2011.

[Lat11b] Chris Lattner. What Every C Programmer Should Know About Undefined Behavior 2/3. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html, May 2011.

[Lat11c] Chris Lattner. What Every C Programmer Should Know About Undefined Behavior 3/3. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html, May 2011.

[LKS$^+$17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming undefined behavior in llvm. *ACM SIGPLAN Notices*, 52(6):633–647, 2017.

[PCL] PC-Lint. https://www.gimpel.com/html/pcl.htm.

[Reg10a] John Regehr. A Guide to Undefined Behavior in C and C++, Part 1. https://blog.regehr.org/archives/213, July 2010.

[Reg10b] John Regehr. A Guide to Undefined Behavior in C and C++, Part 2. https://blog.regehr.org/archives/226, July 2010.

[Reg12] John Regehr. Contest: Craziest Compiler Output due to Undefined Behavior. https://blog.regehr.org/archives/759, July 2012.

[Sea18] Robert Seacord. DCL30-C. Declare objects with appropriate storage durations. https://wiki.sei.cmu.edu/confluence/display/c/DCL30-C.+Declare+objects+with+appropriate+storage+durations, December 2018.

[Seb18a] Martin Sebor. ARR30-C. Do not form or use out-of-bounds pointers or array subscripts. https://wiki.sei.cmu.edu/confluence/display/c/ARR30-C.+Do+not+form+or+use+out-of-bounds+pointers+or+array+subscripts#ARR30-C.Donotformoruseout-of-boundspointersorarraysubscripts-ApparentlyAccessibleOut-of-RangeInde December 2018.

[Seb18b] Martin Sebor. CC. Undefined Behavior. https://wiki.sei.cmu.edu/confluence/display/c/CC.+Undefined+Behavior, July 2018.

[SOL] Solid Sands. https://www.solidsands.com.

[TCC] Tiny C Compiler. https://bellard.org/tcc/.

[UBS] Undefined Behavior Sanitizer, LLVM. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[WCC$^+$12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[WZKSL15] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. A differential approach to undefined behavior detection. *ACM Transactions on Computer Systems (TOCS)*, 33(1):1, 2015.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, pages 283–294. ACM, 2011.

Appendix A

C99 Annex J.2 Undefined behavior

No	Undefined Behavior
1	A“shall” or “shall not” requirement that appears outside of a constraint is violated(clause 4)
2	A nonempty source file does not end in a new-line character which is not immediatelypreceded by a backslash character or ends in a partial preprocessing token orcomment (5.1.1.2)
3	Token concatenation produces a character sequence matching the syntax of auniversal character name (5.1.1.2)
4	A program in a hosted environment does not define a function named main using oneof the specified forms (5.1.2.2.1)
5	A character not in the basic source character set is encountered in a source file, exceptin an identifier, a character constant, a string literal, a header name, a comment, or apreprocessing token that is never converted to a token (5.2.1)
6	An identifier, comment, string literal, character constant, or header name contains aninvalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).
7	The same identifier has both internal and external linkage in the same translation unit(6.2.2).
8	An object is referred to outside of its lifetime (6.2.4)
9	The value of a pointer to an object whose lifetime has ended is used (6.2.42)
10	The value of an object with automatic storage duration is used while it isindeterminate (6.2.4, 6.7.8, 6.8)
11	A trap representation is read by an lvalue expression that does not have character type(6.2.6.1)
12	A trap representation is produced by a side effect that modifies any part of the objectusing an lvalue expression that does not have character type (6.2.6.1).
13	The arguments to certain operators are such that could produce a negative zero result,but the implementation does not support negative zeros (6.2.6.213)
14	Two declarations of the same object or function specify types that are not compatible(6.2.7)
15	Conversion to or from an integer type produces a value outside the range that can berepresented (6.3.1.4)
16	Demotion of one real floating type to another produces a value outside the range thatcan be represented (6.3.1.5)
17	An lvalue does not designate an object when evaluated (6.3.2.1)
18	A non-array lvalue with an incomplete type is used in a context that requires the valueof the designated object (6.3.2.1)
19	An lvalue having array type is converted to a pointer to the initial element of thearray, and the array object has register storage class (6.3.2.1)

20	An attempt is made to use the value of a void expression, or an implicit or explic-itconversion (except to void) is applied to a void expression (6.3.2.2)
21	Conversion of a pointer to an integer type produces a value outside the range that canbe represented (6.3.2.3)
22	Conversion between two pointer types produces a result that is incorrectly aligned(6.3.2.3)
23	A pointer is used to call a function whose type is not compatible with the pointed-totype (6.3.2.3)
24	An unmatched ' or " character is encountered on a logical source line duringtok-enization (6.4)
25	A reserved keyword token is used in translation phase 7 or 8 for some purpose otherthan as a keyword (6.4.1)
26	A universal character name in an identifier does not designate a character whoseen-coding falls into one of the specified ranges (6.4.2.1)
27	The initial character of an identifier is a universal character name designating a digit(6.4.2.1)
28	Two identifiers differ only in nonsignificant characters (6.4.2.1)
29	The identifier _ _func_ _ is explicitly declared (6.4.2.2)
30	The program attempts to modify a string literal (6.4.5)
31	The characters ', \, ", //, or /* occur in the sequence between the <and >delimiters, or the characters ', \, //, or /* occur in the sequence between the "delimiters, in a header name preprocessing token (6.4.7)
32	Between two sequence points, an object is modified more than once, or is modi-fiedand the prior value is read other than to determine the value to be stored (6.5)
33	An exceptional condition occurs during the evaluation of an expression (6.5)
34	An object has its stored value accessed other than by an lvalue of an allowable type(6.5)
35	An attempt is made to modify the result of a function call, a conditional opera-tor, anassignment operator, or a comma operator, or to access it after the next sequencepoint (6.5.2.2, 6.5.15, 6.5.16, 6.5.17)
36	For a call to a function without a function prototype in scope, the number ofargu-ments does not equal the number of parameters (6.5.2.2)
37	For call to a function without a function prototype in scope where the function isdefined with a function prototype, either the prototype ends with an ellipsis or thetypes of the arguments after promotion are not compatible with the types of theparameters (6.5.2.2)
38	For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2)
39	A function is defined with a type that is not compatible with the type (of theex-pression) pointed to by the expression that denotes the called function (6.5.2.2)
40	The operand of the unary * operator has an invalid value (6.5.3.2)
41	A pointer is converted to other than an integer or pointer type (6.5.4)
42	The value of the second operand of the / or % operator is zero (6.5.5)
43	Addition or subtraction of a pointer into, or just beyond, an array object and aninteger type produces a result that does not point into, or just beyond, the same arrayobject (6.5.6)
44	Addition or subtraction of a pointer into, or just beyond, an array object and aninteger type produces a result that points just beyond the array object and is used asthe operand of a unary * operator that is evaluated (6.5.6)

45 Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6)

46 An array subscript is out of range, even if an object is apparently accessible with thegiven subscript (as in the lvalue expression a[1][7] given the declaration inta[4][5]) (6.5.6)

47 The result of subtracting two pointers is not representable in an object of typeptrdiff_t (6.5.6)

48 An expression is shifted by a negative number or by an amount greater than or equalto the width of the promoted expression (6.5.7)

49 An expression having signed promoted type is left-shifted and either the value of theexpression is negative or the result of shifting would be not be representable in thepromoted type (6.5.7)

50 Pointers that do not point to the same aggregate or union (nor just beyond the samearray object) are compared using relational operators (6.5.8)

51 An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1)

52 An expression that is required to be an integer constant expression does not have aninteger type; has operands that are not integer constants, enumeration constants,character constants, sizeof expressions whose results are integer constants, orimmediately-cast floating constants; or contains casts (outside operands to sizeofoperators) other than conversions of arithmetic types to integer types (6.6)

53 A constant expression in an initializer is not, or does not evaluate to, one of thefollowing: an arithmetic constant expression, a null pointer constant, an addressconstant, or an address constant for an object type plus or minus an integer constantexpression (6.6)

54 An arithmetic constant expression does not have arithmetic type; has operands thatare not integer constants, floating constants, enumeration constants, characterconstants, or sizeof expressions; or contains casts (outside operands to sizeofoperators) other than conversions of arithmetic types to arithmetic types (6.6)

55 The value of an object is accessed by an array-subscript [ ], member-access . or ->,address &, or indirection * operator or a pointer cast in creating an address constant(6.6)

56 An identifier for an object is declared with no linkage and the type of the object isincomplete after its declarator, or after its init-declarator if it has an initializer (6.7)

57 A function is declared at block scope with an explicit storage-class specifier otherthan extern (6.7.1)

58 A structure or union is defined as containing no named members (6.7.2.1)

59 An attempt is made to access, or generate a pointer to just past, a flexible arraymember of a structure when the referenced object provides no elements for that array(6.7.2.1)

60 When the complete type is needed, an incomplete structure or union type is notcompleted in the same scope by another declaration of the tag that defines the content(6.7.2.3)

61 An attempt is made to modify an object defined with a const-qualified type throughuse of an lvalue with non-const-qualified type (6.7.3)

62 An attempt is made to refer to an object defined with a volatile-qualified type throughuse of an lvalue with non-volatile-qualified type (6.7.3)

63 The specification of a function type includes any type qualifiers (6.7.3)

64 Two qualified types that are required to be compatible do not have the identicallyqualified version of a compatible type (6.7.3)

65 An object which has been modified is accessed through a restrict-qualified pointer toa const-qualified type, or through a restrict-qualified pointer and another pointer thatare not both based on the same object (6.7.3.1)

66	A restrict-qualified pointer is assigned a value based on another restricted pointerwhose associated block neither began execution before the block associated with thispointer, nor ended before the assignment (6.7.3.1)
67	A function with external linkage is declared with an inline function specifier, but isnot also defined in the same translation unit (6.7.4)
68	Two pointer types that are required to be compatible are not identically qualified, orare not pointers to compatible types (6.7.5.1)
69	The size expression in an array declaration is not a constant expression and evaluatesat program execution time to a nonpositive value (6.7.5.2)
70	In a context requiring two array types to be compatible, they do not have compatibleelement types, or their size specifiers evaluate to unequal values (6.7.5.2)
71	A declaration of an array parameter includes the keyword static within the [and] and the corresponding argument does not provide access to the first element of anarray with at least the specified number of elements (6.7.5.3)
72	A storage-class specifier or type qualifier modifies the keyword void as a function-parameter type list (6.7.5.3)
73	In a context requiring two function types to be compatible, they do not havecompatible return types, or their parameters disagree in use of the ellipsis terminatoror the number and type of parameters (after default argument promotion, when thereis no parameter type list or when one type is specified by a function definition with an identifier list) (6.7.5.3)
74	The value of an unnamed member of a structure or union is used (6.7.8)
75	The initializer for a scalar is neither a single expression nor a single expressionenclosed in braces (6.7.8)
76	The initializer for a structure or union object that has automatic storage duration isneither an initializer list nor a single expression that has compatible structure or uniontype (6.7.8)
77	The initializer for an aggregate or union, other than an array initialized by a stringliteral, is not a brace-enclosed list of initializers for its elements or members (6.7.8)
78	An identifier with external linkage is used, but in the program there does not existexactly one external definition for the identifier, or the identifier is not used and thereexist multiple external definitions for the identifier (6.9)
79	A function definition includes an identifier list, but the types of the parameters are notdeclared in a following declaration list (6.9.1)
80	An adjusted parameter type in a function definition is not an object type (6.9.1)
81	A function that accepts a variable number of arguments is defined without aparameter type list that ends with the ellipsis notation (6.9.1)
82	The } that terminates a function is reached, and the value of the function call is usedby the caller (6.9.1)
83	An identifier for an object with internal linkage and an incomplete type is declared-with a tentative definition (6.9.2)
84	The token defined is generated during the expansion of a #if or #elifpreprocessing directive, or the use of the defined unary operator does not matchone of the two specified forms prior to macro replacement (6.10.1)
85	The #include preprocessing directive that results after expansion does not matchone of the two header name forms (6.10.2)
86	The character sequence in an #include preprocessing directive does not start with aletter (6.10.2)
87	There are sequences of preprocessing tokens within the list of macro arguments thatwould otherwise act as preprocessing directives (6.10.3)
88	The result of the preprocessing operator # is not a valid character string literal(6.10.3.2)

89	The result of the preprocessing operator ## is not a valid preprocessing token(6.10.3.3)
90	The #line preprocessing directive that results after expansion does not match one ofthe two well-defined forms, or its digit sequence specifies zero or a number greaterthan 2147483647 (6.10.4)
91	A non-STDC #pragma preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6)
92	A #pragma STDC preprocessing directive does not match one of the well-definedforms (6.10.6)
93	The name of a predefined macro, or the identifier defined, is the subject of a#define or #undef preprocessing directive (6.10.8)
94	An attempt is made to copy an object to an overlapping object by use of a library-function, other than as explicitly allowed (e.g., memmove) (clause 7)
95	A file with the same name as one of the standard headers, not provided as part of theimplementation, is placed in any of the standard places that are searched for includedsource files (7.1.2)
96	A header is included within an external declaration or definition (7.1.2)
97	A function, object, type, or macro that is specified as being declared or defined bysome standard header is used before any header that declares or defines it is included(7.1.2)
98	A standard header is included while a macro is defined with the same name as a keyword (7.1.2
99	The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2)
100	The program declares or defines a reserved identifier, other than as allowed by 7.1.4) (7.1.3)
101	The program removes the definition of a macro whose name begins with anunderscore and either an uppercase letter or another underscore (7.1.3)
102	An argument to a library function has an invalid value or a type not expected by afunction with variable number of arguments (7.1.4)
103	The pointer passed to a library function array parameter does not have a value suchthat all address computations and object accesses are valid (7.1.4)
104	The macro definition of assert is suppressed in order to access an actual function(7.2)
105	The argument to the assert macro does not have a scalar type (7.2)
106	The CX_LIMITED_RANGE, FENV_ACCESS, or FP_CONTRACT pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2)
107	The value of an argument to a character handling function is neither equal to the valueof EOF nor representable as an unsigned char (7.4)
108	Amacro definition of errno is suppressed in order to access an actual object, or theprogram defines an identifier with the name errno (7.5)
109	Part of the program tests floating-point status flags, sets floating-point control modes,or runs under non-default mode settings, but was translated with the state for theFENV_ACCESS pragma “off” (7.6.1)
110	The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.2)
111	The fesetexceptflag function is used to set floating-point status flags that were not specified in the call to the fegetexceptflag function that provided the value of the corresponding fexcept_t object (7.6.2.4)
112	The argument to fesetenv or feupdateenv is neither an object set by a call to fegetenv or feholdexcept, nor is it an environment macro (7.6.4.3, 7.6.4.4)

113	The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.20.6.1, 7.20.6.2, 7.20.1)
114	The program modifies the string pointed to by the value returned by the setlocale function (7.11.1.1)
115	The program modifies the structure pointed to by the value returned by the localeconv function (7.11.2.1)
116	A macro definition of math_errhandling is suppressed or the program definesan identifier with the name math_errhandling (7.12)
117	An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14)
118	Amacro definition of setjmp is suppressed in order to access an actual function, orthe program defines an external identifier with the name setjmp (7.13)
119	An inv ocation of the setjmp macro occurs other than in an allowed context (7.13.2.1)
120	The longjmp function is invoked to restore a nonexistent environment (7.13.2.1)
121	After a longjmp, there is an attempt to access the value of an object of automatic-storage class with non-volatile-qualified type, local to the function containing thein-vocation of the corresponding setjmp macro, that was changed between thesetjmp invocation and longjmp call (7.13.2.1)
122	The program specifies an invalid pointer to a signal handler function (7.14.1.1)
123	A signal handler returns when the signal corresponded to a computational excep-tion(7.14.1.1)
124	A signal occurs as the result of calling the abort or raise function, and the signal-handler calls the raise function (7.14.1.1)
125	A signal occurs other than as the result of calling the abort or raise function, andthe signal handler refers to an object with static storage duration other than byassigning a value to an object declared as volatile sig_atomic_t, or calls anyfunction in the standard library other than the abort function, the _Exit function,or the signal function (for the same signal number) (7.14.1.1)
126	The value of errno is referred to after a signal occurred other than as the result of calling the abort or raise function and the corresponding signal handler obtained a SIG_ERR return from a call to the signal function (7.14.1.1)
127	A signal is generated by an asynchronous signal handler (7.14.1.1)
128	A function with a variable number of arguments attempts to access its varyingargu-ments other than through a properly declared and initialized va_list object, orbefore the va_start macro is invoked (7.15, 7.15.1.1, 7.15.1.4)
129	The macro va_arg is invoked using the parameter ap that was passed to a function that invoked the macro va_arg with the same parameter (7.15)
130	Amacro definition of va_start, va_arg, va_copy, or va_end is suppressed in order to access an actual function, or the program defines an external identifier with the name va_copy or va_end (7.15.1)
131	The va_start or va_copy macro is invoked without a corresponding invocation of the va_end macro in the same function, or vice versa (7.15.1, 7.15.1.2, 7.15.1.3, 7.15.1.4)
132	The type parameter to the va_arg macro is not such that a pointer to an object ofthat type can be obtained simply by postfixing a * (7.15.1.1)
133	The va_arg macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.15.1.1)

134	The va_copy or va_start macro is called to initialize a va_list that was previously initialized by either macro without an intervening invocation of the va_end macro for the same va_list (7.15.1.2, 7.15.1.4)
135	The parameter parmN of a va_start macro is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.15.1.4)
136	The member designator parameter of an offsetof macro is an invalid rightoperand of the . operator for the type parameter, or designates a bit-field (7.17)
137	The argument in an instance of one of the integer-constant macros is not a decimal,octal, or hexadecimal constant, or it has a value that exceeds the limits for thecorresponding type (7.18.4)
138	Abyte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.19.2)
139	Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.19.2)
140	The value of a pointer to a FILE object is used after the associated file is closed (7.19.3)
141	The stream for the fflush function points to an input stream or to an update stream in which the most recent operation was input (7.19.5.2)
142	The string pointed to by the mode argument in a call to the fopen function does not exactly match one of the specified character sequences (7.19.5.3)
143	An output operation on an update stream is followed by an input operation without an intervening call to the fflush function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.19.5.3)
144	An attempt is made to use the contents of the array that was supplied in a call to the setvbuf function (7.19.5.6)
145	There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2)
146	The format in a call to one of the formatted input/output functions or to the strftime or wcsftime function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.19.6.1, 7.19.6.2, 7.23.3.5, 7.24.2.1, 7.24.2.2, 7.24.5.1)
147	In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.19.6.1, 7.24.2.1)
148	A conversion specification for a formatted output function uses an asterisk to denotean argument-supplied field width or precision, but the corresponding argument is notprovided (7.19.6.1, 7.24.2.1)
149	A conversion specification for a formatted output function uses a # or 0 flag with aconversion specifier other than those described (7.19.6.1, 7.24.2.1)
150	A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2)
151	An s conversion specifier is encountered by one of the formatted output functions,and the argument is missing the null terminator (unless a precision is specified thatdoes not require null termination) (7.19.6.1, 7.24.2.1)
152	An n conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2)

153	A % conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly %% (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2)./
154	An inv alid conversion specification is found in the format for one of the formatted input/output functions, or the strftime or wcsftime function (7.19.6.1, 7.19.6.2, 7.23.3.5, 7.24.2.1, 7.24.2.2, 7.24.5.1)
155	The number of characters transmitted by a formatted output function is greater than INT_MAX (7.19.6.1, 7.19.6.3, 7.19.6.8, 7.19.6.10)
156	The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.19.6.2, 7.24.2.2)
157	A c, s, or [conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is s or [) (7.19.6.2, 7.24.2.2)
158	A c, s, or [conversion specifier with an l qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.19.6.2, 7.24.2.2)
159	The input item for a %p conversion by one of the formatted input functions is not avalue converted earlier during the same program execution (7.19.6.2, 7.24.2.2)
160	The vfprintf, vfscanf, vprintf, vscanf, vsnprintf, vsprintf, vsscanf, vfwprintf, vfwscanf, vswprintf, vswscanf, vwprintf, or vwscanf function is called with an improperly initialized va_list argument, or the argument is used (other than in an invocation of va_end) after the function returns (7.19.6.8, 7.19.6.9, 7.19.6.10, 7.19.6.11, 7.19.6.12, 7.19.6.13, 7.19.6.14, 7.24.2.5, 7.24.2.6, 7.24.2.7, 7.24.2.8, 7.24.2.9, 7.24.2.10)
161	The contents of the array supplied in a call to the fgets, gets, or fgetws function are used after a read error occurred (7.19.7.2, 7.19.7.7, 7.24.3.2)
162	The file position indicator for a binary stream is used after a call to the ungetc function where its value was zero before the call (7.19.7.11)
163	The file position indicator for a stream is used after an error occurred during a call to the fread or fwrite function (7.19.8.1, 7.19.8.2)
164	A partial element read by a call to the fread function is used (7.19.8.1)
165	The fseek function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the ftell function on a stream associated with the same file or whence is not SEEK_SET (7.19.9.2)
166	The fsetpos function is called to set a position that was not returned by a previous successful call to the fgetpos function on a stream associated with the same file (7.19.9.3)
167	A non-null pointer returned by a call to the calloc, malloc, or realloc functionwith a zero requested size is used to access an object (7.20.3)
168	The value of a pointer that refers to space deallocated by a call to the free orrealloc function is used (7.20.3)
169	The pointer argument to the free or realloc function does not match a pointerearlier returned by calloc, malloc, or realloc, or the space has beendeallocated by a call to free or realloc (7.20.3.2, 7.20.3.4)
170	The value of the object allocated by the malloc function is used (7.20.3.3)
171	The value of any bytes in a new object allocated by the realloc function beyondthe size of the old object are used (7.20.3.4)
172	The program executes more than one call to the exit function (7.20.4.32

173 During the call to a function registered with the atexit function, a call is made tothe longjmp function that would terminate the call to the registered function(7.20.4.3)

174 The string set up by the getenv or strerror function is modified by the program(7.20.4.5, 7.21.6.2)

175 A command is executed through the system function in a way that is documented ascausing termination or some other form of undefined behavior (7.20.4.6)

176 A searching or sorting utility function is called with an invalid pointer argument, evenif the number of elements is zero (7.20.5)

177 The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.20.5)

178 The array being searched by the bsearch function does not have its elements in proper order (7.20.5.1)

179 The current conversion state is used by a multibyte/wide character conversion function after changing the LC_CTYPE category (7.20.7)

180 A string or wide string utility function is instructed to access an array beyond the endof an object (7.21.1, 7.24.4)

181 A string or wide string utility function is called with an invalid pointer argument, evenif the length is zero (7.21.1, 7.24.4)

182 The contents of the destination array are used after a call to the strxfrm, strftime, wcsxfrm, or wcsftime function in which the specified length was too small to hold the entire null-terminated result (7.21.4.5, 7.23.3.5, 7.24.4.4.4, 7.24.5.1)

183 The first argument in the very first call to the strtok or wcstok is a null pointer(7.21.5.8, 7.24.4.5.7)

184 The type of an argument to a type-generic macro is not compatible with the type ofthe corresponding parameter of the selected function (7.22)

185 A complex argument is supplied for a generic parameter of a type-generic macro thathas no corresponding complex function (7.22)

186 The argument corresponding to an s specifier without an l qualifier in a call to the fwprintf function does not point to a valid multibyte character sequence that begins in the initial shift state (7.24.2.11)

187 In a call to the wcstok function, the object pointed to by ptr does not have the value stored by the previous call for the same wide string (7.24.4.5.7)

188 An mbstate_t object is used inappropriately (7.24.6)

189 The value of an argument of type wint_t to a wide character classification or casemapping function is neither equal to the value of WEOF nor representable as awchar_t (7.25.15)

190 The iswctype function is called using a different LC_CTYPE category from the one in effect for the call to the wctype function that returned the description (7.25.2.2.1)

191 The towctrans function is called using a different LC_CTYPE category from the one in effect for the call to the wctrans function that returned the description (7.25.3.2.1)