UNIVERSITY OF MOLISE

DEPARTMENT OF BIOSCIENCES AND TERRITORY

MASTER DEGREE COURSE IN SOFTWARE SYSTEM SECURITY

THESIS

IN

SOFTWARE RELIABILITY AND TESTING

An Empirical Study on Bounded Model Checking in Security Context

Author
Stefano DALLA PALMA
Matriculation ID 158350

Advisor
Dr. Rocco OLIVETO

Co-advisor
Dr. Earl BARR

ACADEMIC YEAR 2017-2018

Abstract

Correctness of computer systems is critical in today's information society. Vulnerabilities in programs implementing security features can be catastrophic: for example they may be exploited by malign users to gain access to sensitive data and pose security, privacy, and trust threats thus harming the affiliated users. Program testing is widely used to find bugs in those systems, before the code ever get deployed. However, while testing can show the presence of bugs, it cannot prove their absence. In recent years, automatic software verification has emerged as a complementary approach to program testing for enhancing software quality and new tools have been created and used for years. Model Checking is an automatic verification technique that can guarantee the absence of bugs with respect to a given formal specification and satisfy some desired property the system should have. This thesis examines the ability of Bounded Model Checking to identify vulnerable code regions through a study of *CBMC*, a popular and well-cited tool for the formal verification of ANSI-C programs using Bounded Model Checking. We provide a quantitative and qualitative analysis with the purpose of understanding (i) the real capabilities of existing tools relying on BMC technique in the detection of vulnerable code regions and (ii) how the presence of multiple vulnerabilities in the code can affect the detection of a single vulnerability of interest.

Our findings indicate that even though the distribution of almost all properties (that the model checker must satisfy) among vulnerabilities is significantly different between each other, only two properties are able to strongly discriminate between the vulnerability they are supposed to detect and other vulnerability types. Furthermore, the presence of more vulnerabilities in the source code can affect the detection of some vulnerabilities of interest with respect to almost all the properties considered.

The perspective of this work is of both researchers and practitioners: the former are interested in understanding possible limitations of current approaches in order to devise better ones; the latter - in evaluating the applicability of vulnerable code detection in practice. A sad truth is that in most projects verification is pushed to the very end of the development cycle. It is highly recommendable to use formal verification techniques as early as possible in the development cycle, above all for safety critical systems. The correct implementation of such techniques determines the correctness of the system relying on them. Therefore, strong attention should be posed on them.

Acknowledgements

I would like to acknowledge my advisors, Dr. Rocco Oliveto and Dr. Earl T. Barr, for having followed me during this work. Thanks to Dr. Nassim Seghir for having assisted me during my journey at University College London working on this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

BMC	**B**ounded **M**odel **C**hecking
CNF	**C**onjunctive **N**ormal **F**orm
CVE	**C**ommon **V**ulnerability and **E**xposure
CWE	**C**ommon **W**eaknesses **E**numeration
SAT	**SAT**isfiability

Chapter 1

Introduction

1.1 Application Context

Correctness of computer systems is critical in today's information society. The complexity of the software we use every day has risen dramatically, and correctness of such software is often a big problem. There are more and more devices everywhere connected to the Internet, that are changing the way we work and live by saving time and resources, and opening new opportunities for growth, innovation, and the exchange of knowledge between entities. However, the existence of such a large network of interconnected entities will pose new security, privacy, and trust threats that put all those devices at a high risk, thus harming the affiliated users. Although a lot of companies state that their technologies are secured and protected, they are still prone to various types of attacks. Since the interconnected devices have a direct impact on the lives of users, there is a need for a well-defined security threat classification and a proper security infrastructure with new systems and protocols that can mitigate the security challenges regarding privacy, data integrity, and availability.

Software has bugs, and it seems to be unavoidable. Even though the software development process has evolved a lot, the vulnerabilities mapping back to the same weaknesses. When these vulnerabilities are present in systems controlling critical infrastructure, the consequences can be catastrophic. To find them we often use testing and code reviews. Program testing is widely used to check conformance between software requirements and their implementation. While testing can show the presence of bugs, it cannot prove their absence (Dijkstra, 1969). However, formal methods have emerged as a complementary approach to testing as they enable rigorous reasoning about the software. *Model Checking* (Baier and Katoen, 2008) is an automatic technique for system verification based on exhaustive exploration of all system states that can guarantee the absence of bugs with respect to a given formal specification. Such approaches are needed for safety critical software, where cost of errors is high and crashes can result in loss of human lives. History is full of stories where software errors led to huge money losses or human lives. On its mission to Mars in 1998 the Climate Orbiter spacecraft was ultimately lost in space. Although the failure bemused engineers for some time it was revealed that a sub contractor on the engineering team failed to code a simple conversion from English units to

metric. An embarrassing lapse that sent the $125 million craft fatally close to Mars' surface after attempting to stabilize its orbit too low. Flight controllers believe the spacecraft ploughed into Mars' atmosphere where the associated stresses crippled its communications, leaving it hurtling on through space in an orbit around the sun[1]. The Boeing 787 Dreamliner experienced an integer overflow bug which could shut down all electrical generators if the aircraft was on for more than 248 days[2]. Other examples of software errors are present in the Appendix A.

1.2 Motivation and Objectives

Security is a property of a system that reflects the system ability to protect itself from accidental or deliberate external attacks. It is becoming increasingly important as systems are networked so that external attacks on the system through the Internet are possible. Despite the security community emphasis on the importance of building secure software, security is still not a priority in software development environments and part of developer's mindset while coding (Oliveira et al., 2014). The number of new vulnerabilities discovered so far is increasing. In addition, vulnerabilities that have been studied for years are still commonly reported in vulnerability databases. The Common Vulnerabilities and Exposure[3] (CVE) and the Common Weakness Enumeration[4] (CWE) databases operated by MITRE track the most serious vulnerabilities. At the time of writing, over one hundred thousands CVEs and over seven hundred CWEs were registered.

Most of these vulnerabilities reside in large complex software and manual inspection of these software is infeasible and costly, so automatic support is required. Many tools rely on engineers to provide test-vectors to uncover design flaws. Thus, the process of finding and countering bugs, hacks, and other cyber infection vectors is still human dependent. Formal verification, on the other hand, can be automated, and tools that implement it can check the behavior of a design for any vector of inputs. However, some verification methods require a lot of effort and knowledge from the user to be put into formal proof of correctness, using general purpose theorem provers. Other methods, most notably ones related to model checking, require to build a model of a system (e.g., a state machine) to be proved, which is both labor-intensive and error prone. Nevertheless, one particularly way is to give up trying to prove full correctness of a program, as the associated cost is high. Instead one may attempt to check certain properties of a program, which are known to cause security problems.

[1] `https://mars.jpl.nasa.gov/msp98/news/mco991110.html` (accessed on June, 2018)

[2] `https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf` (accessed on June, 2018)

[3] The MITRE CVE system provides a reference-method for publicly known information-security vulnerabilities and exposures and can be found at: `https://cve.mitre.org/`

[4] The MITRE CWE is a community-developed dictionary of software weakness types and can be found at: `https://cwe.mitre.org/`

While there are a number of papers related to verification or checking of C code (Ball and Rajamani, 2002; Agten, Jacobs, and Piessens, 2015; Eisner, 2005; Dahlweid et al., 2009; Filliâtre and Marché, 2004; Holzmann, 2000) as well as static analysis tools comparison, very few tools exist which are capable of advanced reasoning about ANSI-C/C++ code. Since its development in the early 1970s, the C programming language has become one of the most widely used programming languages of all time and still remains nowadays[5]. Many today's applications - including operating systems, database management systems and embedded systems (which appear in cars, airplanes and industrial control systems) - are still tipically implemented in C because of its efficiency and access to low-level features. However, the features that make C desiderable for many system-level programming tasks - namely its weak typing, low-level access to computer memory, and pointers - put the systems that implement it at high risk of vulnerability. These systems are all of critical importance. They are the platform for computing, and they drive our economy, and ourselves. A successful attack on these systems can have tremendous consequences.

This work was guided by the motivation to understand (i) the real capabilities of existing tools relying on the Bounded Model Checking (Biere et al., 1999) technique in the detection of vulnerable code regions and (ii) how the presence of multiple vulnerabilities in the code can affect the detection of a single vulnerability of interest. The research aims at answering the following research questions:

- **RQ1.** *What is the distribution of the safety properties among known vulnerabilities?*
- **RQ2.** *To what extent does the distribution of properties and the presence of different vulnerabilities affect the detection of a given vulnerability?*
- **RQ3.** *Which are optimal values of loop unwinds for which precision and recall are maximized?*

The perspective is of both researchers and practitioners: the former are interested in understanding possible limitations of current approaches in order to devise better ones; the latter - in evaluating the applicability of vulnerable code detection in practice.

1.3 Results

The result of this work is an empirical study on Bounded Model Checking in security context through the study of *CBMC* (Clarke, Kroening, and Lerda, 2004), a popular and well-cited open-source tool for the formal verification of ANSI-C programs using BMC. *CBMC* is able to check whether a property is always valid for any

[5] C is currently the second most used programming language. For decades, Tiobe has generated an index of the most popular programming languages. They update this list monthly, pulling in data from hundreds of sources around the world. The index can be found at `https://www.tiobe.com/tiobe-index/` (accessed on July, 2018)

execution of the program. It verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. It model integer arithmetic accurately, and is able to reason about machine-level artifacts such as integer overflow. In particular, *CBMC* is unique in its ability to support almost all features of C language and to handle well constructions as arithmetics with bounded integers, pointer arithmetics, bitwise operations and function pointers. Therefore, it is able to detect a class of bugs that has so far gone unnoticed by many formal verification tools.

The study consisted of the analysis of 5.355 functions, taken from the Juliet Test Suite[6], a collection of test cases in the C/C++ language that contains examples organized under 118 different CWEs.

With respect to **RQ1** our findings indicate that even though the distribution of almost all properties (that the model checker must satisfy) among vulnerabilities is significantly different between each other, only three properties are able to strongly discriminate between the vulnerability they are supposed to detect and other vulnerability types. Furthermore, in **RQ2** we have seen that the presence of more vulnerabilities in the source code can affect the detection of some vulnerabilities of interest with respect to almost all the properties considered. Finally, in **RQ3**, we found out there is no need to use a high value of loop unwinding in the context of the analysed vulnerabilities. Indeed, there exist several local optima for which precision and recall are maximized using a small unwind value. Of course, the greater the value the lower the number of verifications that terminates within a given timebound.

1.4 Thesis Structure and Organization

This remainder of this thesis is organized as follow:

- Chapter 2 introduces CBMC and describes the related work.
- In Chapter 3 we describe the study aimed at analyzing the properties distribution among several CWEs and to assess the influence of the presence of different vulnerabilities in detecting a single vulnerability of interest. Then, we discuss the results and report possible threats affecting our findings and how we mitigated them.
- Chapter 4 concludes the thesis and outlines our future research agenda.

[6]The test suite is available at `https://samate.nist.gov/SRD/testsuite.php` (accessed on September, 2018)

Chapter 2

Background and Related Work

2.1 Introduction

Our work is related to various topics, including static analysis for security audits and bounded model checking. We will specifically focus on the area of Model Checking, i.e., checking whether a given model satisfies an interesting, predefined property and on the problem of Bounded Model Checking, where the model is bounded in size. Finally, we will then describe the model checking tool *CBMC*, which implement BMC over the C programming language.

2.2 Background

Attackers have been exploiting the many of the same weaknesses in software for decades. The exploits change to suit the platform on specific implementation of a vulnerability, but the underlying weaknes is the same, and the work to exploit a vulnerability is largely the same. These vulnerabilities manifest as runtime errors, which occur while the program is running. The term is often used in contrast to other types of program errors, such as syntax errors and compile time errors. There are many different types of runtime errors. One example is a logic error, which produces the wrong output. For example, a miscalculation in the source code or a spreadsheet program may produce the wrong result when a user enters a formula into a cell. Another type of runtime error is a memory leak. This type of error causes a program to continually use up more RAM while the program is running. A memory leak may be due to an infinite loop, not deallocating unused memory, or other reasons. A program crash is the most noticeable type of runtime error, since the program unexpectedly quits while running. Crashes can be caused by memory leaks or other programming errors. Common examples include dividing by zero, referencing missing files, calling invalid functions, or not handling certain input correctly. Normally, a program with one or more of these bugs will simply crash. But an attacker can alter the situations that cause the program to steal private information, corrupt valuable information or run arbitrary code.

Therefore, the discovery of vulnerabilities in program code is a fundamental problem of computer security. Consequently, it has received much attention in the past. Static analysis of the source code can help eliminate various security bugs.

Static analysis (SA) is a method to reason about runtime properties of program code without executing it. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. In software and hardware design of complex systems, more time and effort are spent on verification than on construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Automated tools can assist programmers and developers in carrying out static analysis, especially during Security Audits.

(IEEE, 1997) defines a software audit as *"An independent examination of a software product, process, or set of software processes to assess compliance with specifications, standards, contractual agreements, or other criteria"*. In theory, security audits should identify and remove the vulnerability before the code even get deployed. While it is considered a best practice to perform code reviews before code is released, there is often not enough manpower or expertise to rigorously review all the code that should be reviewed. In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer (*Clang Static Analyzers*) as well as the dynamic analyzer Valgrind (Nethercote and Seward, 2007) , CBMC (Clarke, Kroening, and Lerda, 2004) and SatAbs (Clarke et al., 2005), and others, can pinpoint further pitfalls such as invalid memory access. Static analysis tools like Flawfinder (*Flawfinder*) and VCCFinder (Perl et al., 2015) help to find possible security vulnerabilities searching through C/C++ source code looking for potential security flaws.

Thus, static code analysis can generate warnings for developers so that code is more correct and bug-free thereby preventing weaknesses in production software, thwarting would-be hackers looking for software exploits. However, static analysis does not in general guaranteee the absence of runtime errors. Formal methods offer a large potential to deal with them.

One approach is model checking, a model-based formal verification technique which consists of a systematically exhaustive exploration of the mathematical model of the system (this is possible for finite models, but also for some infinite models where infinite sets of states can be effectively represented finitely by using abstraction).

2.2.1 Model Checking

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. *Model checking* is a verification technique that explores all possible system states in a brute-force manner (Baier and Katoen, 2008). The term Model Checking was coined by Clarke

and Emerson (Clarke and Emerson, 1982) in the early eighties. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios (i.e., all relevant system states) in a systematic manner. In this way, it can be shown that a given system model M truly satisfies a certain property P: $M \models P$. The model M is usually described by a state machine, while the property P is given by a logic formula over the machine states. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. In general, properties are classified to *safety* and *liveness* properties (Biere et al., 2003). While the former declares what should not happen (or equivalently, what should always happen), the latter declares what should eventually happen. A counterexample to safety properties is a trace of states, where the last state contradicts the property. A counterexample to liveness properties, in its simplest form, is a path to a loop that does not contain the desired state. Such a loop represents an infinite path that never reaches the specified state. Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result ok? Can the system reach a deadlock situation, e.g., when two concurrent programs are waiting for each other and thus halting the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset? or more specific technical properties: Does a given program ever dereference a null pointer? However, Model Checking requires a precise and unambiguous statement of the properties to be examined.

The model M, as well as the state machine which describes it, can either be bounded or infinite. For example, one can describe a computer program P over variables $x_1, \ldots, x_n$, by defining a state machine S in which each state represent a possible combination of assignments for the variables. If the values are computationally bounded, e.g. 32 bit values, then the set of states is bounded as well by $2^{32} \cdot n$. However, if the arithmetic model is unbounded, then so are the values of the variables, and so the set of possible states is infinite. Intuitively, verifying a property P over an infinite model is more complex then doing so over a bounded one. However, even if the given model is known to be bounded, and in some cases even if the bound itself is known (but is very large), the task of verifying a property is not always simple.

2.2.2 Bounded Model Checking

The notion of Bounded Model Checking (BMC) was first intoduced by (Biere et al., 1999), where the bound is the maximal length of a counterexample. In the BMC problem we treat the model, M, as a finite state machine with at most k states for some constant k. In other words, our problem is now: given a finite state machine

M and a property P, either show a counterexample for P with at most k state transitions, or argue there is no such example. The original motivation of bounded model checking was to leverage the success of SAT in solving Boolean formulas to model checking. The great advantage of model checking is that it is often fully automatic; its primary disadvantage is that it does not in general scale to large systems. However, during the last few years there has been a tremendous increase in reasoning power of SAT solvers. Indeed, they can now handle instances with hundreds of thousands of variables and millions of clauses.

2.3 Bounded Model Checker for ANSI-C

CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check ANSI-C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure, an algorithm that, given a decision problem, terminates with a correct yes/no answer. While CBMC is aimed at embedded software, it also supports dynamic memory allocation using malloc and new.

Apart from the basic application of verifying C programs, CBMC is used for a wide variety of applications, such as[1]:

- Error explanation: extended versions of CBMC can find and explain the cause of an error.
- BMC of concurrent programs: verifying C programs executed by multiple threads.
- Equivalence checking: checking whether two programs are equivalent, in the sense that they always compute the same output.
- Verifying embedded programs and non-program models: the models are formalized using C code and verified using CBMC.
- Verifying existing programs, such as Linux and Windows device drivers.
- Worst-case execution time: analyzing the execution time of programs.
- Security: measuring information leakage in programs, finding security bugs and more.

Generally, the CBMC can be described as a fairly simple program: the program is fed into the program analysis engine, along with a claim (property to satisfy) and a

[1] A more comprehensive list of CBMC applications and usage examples can be found at the Applications of CBMC page: https://www.cprover.org/cbmc/applications/ (accessed on June, 2018)

bound k, which defines the maximal "unfolding" done to program loops. The analysis engine then generates a CNF formula which describes the program, along with a term which describes the claim or property. Specifically, it looks for an assignment which satisfies both the problem and the negation of the claim, to show a contradiction of the claim, or prove that no such contradiction exists up to the given execution bound k. The resulting CNF is then fed into a SAT Solver, which either states it is satisfiable (ideally while showing a counterexample which disproves the claim), or argues that it is unsatisfiable, meaning the property is held.

2.3.1 Loop Unwinding

In BMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using an efficient SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. If the formula is not satisfiable, the program can be unwound more to determine if a longer counterexample exists. In many engineering domains, real-time guarantees are a strict requirement. An example is software embedded in automotive controllers. Therefore, the loop constructs in these types of programs often have a strict bound on the number of iterations.

CBMC can formally verify such bounds by means of unwinding assertions. Once this bound is established, CBMC can prove the absence of errors. The basic idea of CBMC is to model the computation of the programs up to a particular depth k. Technically, this is achieved by a process that essentially amounts to unwinding loops. This concept is best illustrated with a generic example[2]:

```
int main(int argc, char **argv) {
  while(cond) {
    Body;
  }
}
```

After k-unwindings, the `while` loop is either removed or replaced by an unwinding assertion. The unwinding assertion is used to verify that the unwinding was sufficient, i.e. that the loop cannot be executed more then k times (where k is the input bound). A BMC instance that will find bugs with up to k=3 iterations of the loop would contain three copies of the loop body, and essentially corresponds to checking the following loop-free program:

```
int main(int argc, char **argv) {
  if(cond) {     // first unwind
    Body;
    if(cond) {     // second unwind
      Body;
      if(cond) {     // third unwind
        Body;
```

[2]http://www.cprover.org/cprover-manual/cbmc-loops.shtml (accessed on June, 2018)

```
          assert(!cond);   // unwinding assertion
        }
      }
    }
}
```

LISTING 2.1: Iteration-based Unwinding

This results in changes either in the soundness or in the correctness of the verification. Assume k was used as an unwinding bound, and the loop was infact executed $k_0 > k$ times, meaning the unwinding was insufficient. Then:

- If an unwinding assertion was used, an error will be wrongly reported: the loop was executed more then k times, but this does not necessarily mean that an error occurred.
- If an unwinding assertion was not used, and an error might happen in the k + 1-th loop entry, then the error was not reported.

The construction above is meant to produce a program that is trace equivalent with the original programs for those traces that contain up to three iterations of the loop.

In many cases, CBMC is able to automatically determine an upper bound on the number of loop iterations. This may even work when the number of loop unwindings is not constant. Furthermore, CBMC allows defining different unwind bounds for different program loops, allowing the analysis to be more efficient, but requiring more work from the programmer.

2.3.2 Built-in Properties

CBMC uses assertions to specify program properties. Assertions are claims about the state of the program when the program reaches a particular program location. Assertions are often written by the programmer by means of the assert macro. In addition to the assertions written by the programmer, assertions for specific properties can also be generated automatically by CBMC, often relieving the programmer from writing "obvious" assertions.

CBMC comes with an assertion generator called goto-instrument, which performs a conservative static analysis to determine program locations that potentially contain a bug. Due to the imprecision of the static analysis, it is important to emphasize that these generated assertions are only potential bugs; the Model Checker first needs to confirm that they are indeed real bugs. The assertion generator can generate assertions for the verification of the following properties:

- **Buffer overflows**. For each array access, check whether the upper and lower bounds are violated.
- **Pointer safety**. Search for NULL-pointer dereferences or dereferences of other invalid pointers.

- **Division by zero**. Check whether there is a division by zero in the program.
- **Not-a-Number**. Check whether floating-point computation may result in NaNs.

CBMC attempts to build counterexamples that refute the property. If such a counterexample is found, it is presented to the engineer to facilitate localization and repair of the program. Some examples of programs on which CBMC refutes the properties are listed below.

```
void main(){
    char string[10];
    int i;

    for(i=0; i<=10; i++)
        string[i] = '\0';
}
```

LISTING 2.2: Array upper bound access during initialization of variable 'string'

In listing 2.2 and upper bound access occurs in the array `string`, because of the wrong condition `i<=10`. CBMC reports the following messages that allow to identify the bug:

```
[main.array_bounds.1] array 'string' lower bound in string[(int)i]:  SUCCESS
[main.array_bounds.2] array 'string' upper bound in string[(int)i]:  FAILURE
```

```
typedef struct bar{
    char *string;
}Bar;

void foo(Bar *bar){
    bar->string = "Hello World!";
}
```

LISTING 2.3: Null Pointer Dereference

In listing 2.3 the function `foo` is trying to dereference a pointer to a structure `Bar` that could be NULL, leading to a NULL Pointer Dereference. CBMC identifies this kind of bug and reports the following messages:

```
[foo.pointer_dereference.1] dereference failure:  pointer NULL in bar:  FAILURE
[foo.pointer_dereference.2] dereference failure:  pointer invalid in bar:
FAILURE
[foo.pointer_dereference.3] dereference failure:  deallocated dynamic object
in bar:  FAILURE
[foo.pointer_dereference.4] dereference failure:  dead object in bar:  FAILURE
[foo.pointer_dereference.5] dereference failure:  pointer outside dynamic
object bounds in bar:  FAILURE
```

```
[foo.pointer_dereference.6] dereference failure: pointer outside object
bounds in bar: FAILURE
```

Here the keyword `SUCCESS` indicates that CBMC has not been able to find a counterexample for the given property and it assumed that such an example doesn't exist, meaning the property is held. Differently, the keyword `FAILURE` indicates that a counterexample has been found. Hence, we treat messages containing the keyword `FAILURE` as *warnings* provided by CBMC about possible vulnerable statements. We use this warnings in our analysis to verify whether CBMC is able to identify the statements that generate flaws in the analysed vulnerable functions.

Consider the following example:

```
void foo(int a, int b, int c){
    int tmp;
    if (a > b) {tmp = a; a = b; b = tmp;}
    if (b > c) {tmp = b; b = c; c = tmp;}
    if (a < b) {tmp = a; a = b; b = tmp;}
    assert(a<=b && b<=c);
}
```

LISTING 2.4: Counterexample

Looking at this program, one can quite easily find an assignment which violates the assertion. Perhaps the simplest example would be a = 1, b = 1, c = 0. Running CBMC on this example results in the following output:

```
    Counterexample:

State 17 file test.c line 3 thread 0
---------------------------
a=603979795 (00100100000000000000000000010011)
State 18 file test.c line 3 thread 0
---------------------------
b=603979795 (00100100000000000000000000010011)
State 19 file test.c line 3 thread 0
---------------------------
c=603979794 (00100100000000000000000000010010)
State 20 file test.c line 5 function f thread 0
---------------------------
temp=0 (00000000000000000000000000000000)
State 23 file test.c line 7 function f thread 0
---------------------------
temp=603979795 (00100100000000000000000000010011)
State 24 file test.c line 7 function f thread 0
---------------------------
b=603979794 (00100100000000000000000000010010)
```

```
State 25 file test.c line 7 function f thread 0
--------------------------
c=603979795 (00100100000000000000000000010011)

Violated property:
file test.c line 9 function f
assertion a<=b && b<=c
FALSE
VERIFICATION FAILED
```

Indeed, a counterexample is found, even though the numbers used for the counterexample are enormous: `a = 603979795`, `b = 603979795`, `c = 603979794`.

2.4 Related Tool that Uses Abstraction

SatAbs is another verification tool from the CProver suite for ANSI-C/C++ programs. Like CBMC, it allows the verification of following properties:

- **Buffer overflows**. For each array, SatAbs checks whether the upper and lower bounds are violated whenever the array is accessed (i.e., whenever the program reads from or writes to the buffer).
- **Pointer safety**. SatAbs searches for null-pointer dereferences.
- **Divison by zero**. SatAbs checks whether there is a path in the program that executes a divison by zero.
- **User specified assertions**. This is the most generic class of supported properties. SatAbs checks for assertion violations. The user can use the assert function to specify arbitrary conditions that have to hold at certain points in the program.

In many cases, lightweight properties such as array bounds do not rely on the entire program. A large fraction of the program is irrelevant to the property. SatAbs exploits this observation and computes an abstraction of the program in order to handle large amounts of code. Even for very trivial C programs it is impossible to exhaustively examine their state space (which is potentially unbounded). However, not all details in a C program necessarily contribute to a bug, so in theory it is sufficient to examine the parts of the program that are somehow related to a bug. In practice, many static verification tools try to achieve this goal by applying heuristics. This approach comes at a cost: bugs might be overlooked because the heuristics do not cover all relevant aspects of the program. Therefore, the conclusion that a program is correct whenever such a static verification tool is unable to find an error is invalid. The idea underlying SatAbs is to generate a sound abstraction of the original program. In this context, soundness refers to the fact that the abstract program

contains (at least) all relevant behaviours (i.e., bugs) that are present in the original program. SatAbs transforms a C/C++ program into a Boolean program, which is an abstraction of the original program in order to handle large amounts of code. The Boolean program is then passed to a model checker. To build the Boolean program, SatAbs uses Predicate Abstraction (Graf and Saïdi, 1997), which is best suited for lightweight properties such as array bounds (buffer overflows), pointer safety, exceptions and control-flow oriented user-specified assertions.

Chapter 3

Empirical study

3.1 Research Problem

The goal of the empirical study was to analyze the *precision* of the Bounded Model Checking technique with respect to the properties distribution between a vulnerability of interest and other vulnerable instances, with the purpose of understanding the real capabilities of existing tools realying on that technique in the detection of vulnerable code regions. The perspective is of both researchers and practitioners: the former are interested in understanding possible limitations of current approaches in order to devise better ones; the latter - in evaluating the applicability of vulnerable code detection in practice.

3.2 Research Questions

We pose the following research questions:

- **RQ1.** *What is the distribution of the safety properties among known vulnerabilities?*
- **RQ2.** *To what extent does the distribution of properties and the presence of different vulnerabilities affect the detection of a given vulnerability?*
- **RQ3.** *Which are optimal values of loop unwinds for which precision and recall are maximized?*

RQ1 can be considered as a sanity check aimed at assessing whether the distribution of the *safety* properties among several vulnerabilities is different. The assumption is that if two or more properties are similarly distributed between different vulnerabilities, then those properties can be used to detect them but they are not able to specialize and distinguish between vulnerability types. Even though it would be fine if properties generalise in such a way to detect multiple types of vulnerability, one can incur in wrong detections if he/she were looking for a specific vulnerability for which those properties have been defined.

In particular, **RQ2** aimed at assessing the performance of BMC technique in a setting where a code region can be affected by multiple vulnerabilities in such a way they can introduce noise and "disturb" the detection of other vulnerabilities. Given a

vulnerability and a set of properties P that allow identifying it, we ask to what extent does the presence of other vulnerabilities in the code influence the performance of the tool, with respect to P, in correctly identifying the vulnerability.

Finally, with **RQ3** we ask which are optimal number of loop unwinds for which both precision and recall are maximized. This is a general question aimed at understanding the role of loop unwinding in the detection of vulnerable code regions.

3.3 The Juliet Test Suite

The source code that we executed the CBMC on are from the Juliet Test Suite[1] for C and C++ code, version 1.3, created by the National Security Agency's (NSA) Center for Assured Software (CAS) and developed specifically for assessing the capabilities of static analysis tools. It is intended for anyone who wishes to use the test cases for their own testing purposes, or who would like to have a greater understanding of how the test cases were created. The test suite contains 64,099 individual test cases under 118 CWEs. Most of them were generated using source files that contain the flaw and a tool called the "Test Case Template Engine" created by the CAS. Some flaw types could not be generated by the CAS's custom Test Case Template Engine. Test cases for those flaw types were manually created.

We used the Juliet Test Suite as it contains examples organized under 118 different CWEs. CWEs are practically useful for our experiments because they provide a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities as they are found in code, design, or system architecture. Each test case, indeed, is part of an individual CWE, which represents a single vulnerability type. An extract of the Common Weakness Enumerations present in the Juliet Test Suite can be found in Table B.1 in Appendix B.

Furthermore, test cases contain both a vulnerable version and a bug-free one. The distinction between the vulnerable and not vulnerable version is identified by one or more instances of a preprocessor directive `#ifdef OMITGOOD` (to define for testing the vulnerable code) or `#ifdef OMITBAD` (to define for testing the bug-free code) followed by an optional `#else` and `#endif`. Listing 3.1 shows an extract of a cwe from the Juliet Test Suite.

This explicit distinction between vulnerable and not vulnerable version allowed us to run the tool on both vulnerable and bug-free code regions, analysing its behavior in a systematic fashion.

[1] The test cases are publically available for download at `http://samate.nist.gov/SRD/testsuite.php` (accessed on September, 2018)

```
//#define OMITGOOD to execute the vulnerable version
//#define OMITBAD to execute the bug-free version

int main(int argc, char * argv[])
{
#ifndef OMITGOOD
    printLine("Calling good()...");
    CWE121__char_type_overrun_memcpy_good();
    printLine("Finished good()");
#endif /* OMITGOOD */
#ifndef OMITBAD
    printLine("Calling bad()...");
    CWE121__char_type_overrun_memcpy_bad();
    printLine("Finished bad()");
#endif /* OMITBAD */
    return 0;
}
```

LISTING 3.1: Extract of a test case about Stack Based Buffer Overflow

3.4 Properties Distribution

RQ1. *What is the distribution of the safety properties among known vulnerabilities?*

3.4.1 Design

To answer our first research question we have analysed the distribution of the safety properties implemented by CBMC on 67 CWEs. For each property we have run the tool on the selected CWEs and counted how many times each of property did not hold in each file representing a vulnerability, i.e., how many times the verifications on each CWE failed because of each property.

3.4.2 Dataset Setup

Among the 118 CWEs in the Juliet Test Suite we have uniformly selected 15 functions per CWEs from 67 CWEs for a total of 1,005 functions. Given the variable number of functions in each CWE, we believe that 15 functions per CWE is a reasonable tradeoff between the number of functions to analyse and the time required by CBMC to run with all its properties activated. Table B.1 in Appendix B shows the CWEs included within the dataset.

Even though the Juliet Test Suite is composed of 118 CWEs, six of them contains two functions in average; seven are related to *.cpp* code; 38 rely on libraries not available on the machine where the experiments have been carried on. Thus, we have ignored them.

3.4.3 Results

To assess whether the distribution of properties is statistically significant among different vulnerabilities and figure out what method to use, we have first assessed the distribution of the failures on the CWEs by means of the Shapiro-test. The Shapiro-test tests the null hypotesis that the samples came from a Normal distribution. This means that if the Shaipro-test's p-value ≤ 0.05, then we would reject the null hypothesis. We set up a significant level of 0.05 and run the test. The value obtained by the test on our dataset had a *p-value < 2.2e-16 < 0.05*. Thus, according to the test, the samples selected for this analysis does not come from a normal distribution. We then proceeded to measure the significance of the differences in the samples population.

Our data contain the results of the application of the tool on 1,005 files representing 67 distinct vulnerability types regarding safety properties. These were divided in 10 groups: *bounds-check*, *conversion-check*, *division-by-zero-check*, *float-overflow-check*, *memory-leak-check*, *nan-check*, *pointer-check*, *pointer-overflow-check*, *signed-overflow-check*, *unsigned-overflow-check*. After doing so for all cwes, the number of failures for each file (i.e., implementation of a specific vulnerability type) were counted (failure score). Figure 3.1 plots failures by group.

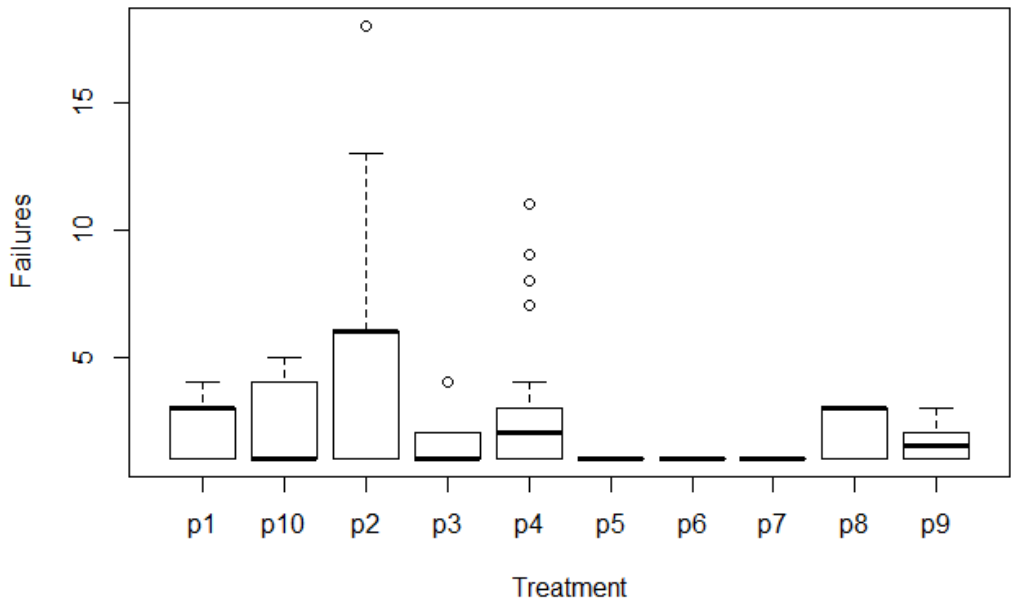

FIGURE 3.1: Failures by group. *p1: bounds-check; p2: pointer-check; p3: unsigned-overflow-check; p4: conversion-check; p5: memory-leak-check; p6: div-by-zero; p7: nan-check; p8: float-overflow-check; p9: pointer-overflow-check; p10: signed-overflow-check*

The very basic research question is:

How do failure scores differ between different properties among vulnerabilities?

That is, we have tested if the ten means - each calculated on a different group of properties - are equal or there exists at least one of the samples (groups) comes from a different population than the others. The most likely test for this scenario is a one-way ANOVA but using it requires some assumptions, among which the one for which the sample distribution must follow a Normal distribution. The checks performed with the Shapiro test told us that these assumptions are not satisfied by our data at hand.

The Kruskal-Wallis H test is a rank-based nonparametric test designed for precisely this situation which does not require a normal distribution that can be used to determine if there are statistically significant differences between two or more groups of an independent variable on a continuous or ordinal dependent variable. It is considered the nonparametric alternative to the one-way ANOVA, and an extension of the Mann-Whitney U test to allow the comparison of more than two independent groups.

We used the Kruskal-Wallis H test to understand whether the number of failures of the tool on different cwes differed based on the property chosen (i.e., the dependent variable was the "number of failures" and the independent variable the chosen

"property", which has ten independent groups, as explained above).

The null hypothesis assumes that the samples (groups) are from identical populations. To test the hypothesis, we applied the Kruskal-Wallis test to compare the "property" data. The *p-value* turned out to be nearly zero: 2.2e-16, with an effect size (e-squared for Kruskall-Wallis test) equals to 0.22. Hence we rejected the null hypothesis. As the p-value is less than the significance level 0.05, we can conclude that there are significant differences between the treatment groups (i.e. properties), that is, the properties failures density in the analysed CWEs are non identical populations.

From the output of the Kruskal-Wallis test, we know that there is a significant difference between groups, i.e. one sample stochastically dominates one other sample. However, the test does not identify where this stochastic dominance occurs or for how many pairs of groups stochastic dominance obtains. For analyzing the specific sample pairs for stochastic dominance in post hoc testing, Dunn's test, pairwise Mann-Whitney tests, or the more powerful but less well known Conover–Iman test are appropriate (Dunn, 1964). It's possible to use the Pairwise Mann–Whitney U-tests to calculate pairwise comparisons between group levels with corrections for multiple testing. To prevent the inflation of type I error rates, adjustments to the p-values have been made by means of Bonferroni correction. Table 3.1 shows the pairs of propety that are significantly different ($p < 0.05$) and their p-values.

Property 1	Property 2	*p-value*
pointer-check	memory-leak	9.16e-13
pointer-check	signed-overflow-check	4.94e-11
conversion-check	memory-leak-check	4.10e-09
pointer-check	unsigned-overflow-check	1.29e-08
bounds-check	memory-leak	3.22e-08
pointer-check	conversion-check	6.97e-08
bounds-check	pointer-check	1.42e-06
pointer-check	div-by-zero-check	1.90e-06
pointer-check	pointer-overflow-check	6.63e-06
pointer-check	float-overflow-check	2.18e-05
conversion-check	div-by-zero-check	1.32e-04
unsigned-overflow-check	memory-leak	2.80e-04
bounds-check	div-by-zero	3.11e-04
memory-leak-check	signed-overflow-check	3.17e-04
conversion-check	nan-check	7.04e-04
bounds-check	nan-check	9.24e-04
memory-leak-check	pointer-overflow-check	7.25e-03
unsigned-overflow-check	div-by-zero	2.45e.02
div-by-zero-check	pointer-overflow-check	2.85e-02
div-by-zero-check	signed-overflow-check	2.95e-02
bounds-check	pointer-over-check	2.95e-02
unsigned-overflow-check	nan-check	3.57e-02

TABLE 3.1: Pairs of property that are significantly different (*p-value* < 0.05) ordered for increasing values of p-value

Summary for RQ1. The properties distribution is significantly different among vulnerability types and they could detect and distinguish vulnerabilities in a more discriminating way than others.

3.5 Performance Evaluation

RQ2. *To what extent does the distribution of properties and the presence of different vulnerabilities affect the detection of a given vulnerability?*

3.5.1 Design

To answer our second research question we have performed the analysis on the performance of the tool on fifteen families of vulnerability (i.e. CWEs). For each CWE we have created a dataset uniformly selecting (i) a set of vulnerabilities belonging to the CWE, for which we have defined a set of properties (available in CBMC) able to identify them; (ii) a set of bug-free functions; and (iii) a set of vulnerable functions belonging to different CWEs.

We have run the tool on two version of the dataset: a version ("pure dataset") containing the vulnerable functions under consideration along with a set of bug-free functions and a version ("injected dataset") in which we added vulnerable functions belonging to different families with respect to the analysed CWE with the purpose of simulating code regions containing more than a single vulnerability. Then, we have evaluated its detection performance, relative to the defined properties, in terms of *precision* on both the starting dataset and the injected one (i.e., before and after injecting the dataset with the vulnerable functions in (iii)). The assumption here is that if the precision decreases considerably after having injected other vulnerabilities in the code, belonging to a different family than the analysed one, then more flaws have been found with the predefined properties and more specific property definitions are needed for discriminatinig more the CWE consiedered from the injected ones. Otherwise, if the precision remains almost similar after the injection, the properties used to detect the specific CWE are not able to detect other types of vulnerability, meaning that those properties are useful in the detection of the considered vulnerability type as they are able to strongly discriminate between vulnerabilities in such a way the detection is not affected by the presence of other vulnerabilties in the source code.

The analysis on the first dataset served us to verify how well the BMC technique distinguish between vulnerable code for which a predefined set properties should not hold (i.e. they allow identify the vulnerability) and not vulnerable code, in an ideal scenario. Remember that, given a property of interest P, we say P is held (up to a given to a given bound k in the case of BMC) if the representing formula fed into the SAT Solver is unsatisfiable. Otherwise we say P is not held.

The analysis on the injected dataset allowed us to verify whether there exists some relationship between the vulnerability considered and vulnerabilities belonging to different families (relative to the predefined set of properties) that can affect its detection.

Finally, for each vulnerability the analysis has been performed bearing in mind the ratio of true positive, false positive, true negative and false negative defined as follows:

- a true positive (TP) is a *pure* vulnerable function identified as vulnerable;
- a false positive (FP) is a bug-free function or an *injected* vulnerable function identified as vulnerable;
- a true negative (TN) is a bug-free function or an *injected* vulnerable function identified as bug-free;
- a false negative (FN) is a *pure* vulnerable function identified as bug-free.

3.5.2 Dataset Setup

From the Juliet Test Suite we have extracted a total of 4,350 functions. In particular, for each of the fifteen CWEs used in our experiment we have uniformly extracted 300 functions, in which 1/3 was composed of vulnerable functions belonging to the CWE considered, 1/3 was composed of bug-free functions and 1/3 about vulnerable functions belonging to a family of vulnerabilities different from the analysed one, used as injected code. Only for CWE 681 we have extracted a total of 150 functions, due to the lack of functions in the test suite for that type of vulnerability.

The fifteen CWEs have been chosen with respect to the available properties implemented in CBMC. That is, we have selected those CWEs for which we knew the properties (or combinatitons of them) do not hold.
Table 3.2 shows the analysed CWEs and their property sets.

CWE ID	Name	Selected properties
CWE121	Stack-based Buffer Overflow	bounds-check, pointer-check, pointer-overflow-check
CWE122	Heap-based Buffer Overflow	bounds-check, pointer-check, pointer-overflow-check
CWE124	Buffer Underwrite	bounds-check, pointer-check, pointer-overflow-check
CWE126	Buffer Overread	bounds-check, pointer-check, pointer-overflow-check
CWE127	Buffer Underread	bounds-check, pointer-check, pointer-overflow-check
CWE190	Integer Overflow and Wraparound	signed-overflow-check, unsigned-overflow-check
CWE191	Integer Underflow	signed-overflow-check, unsigned-overflow-check
CWE194	Unexpected Sign Extension	conversion-check
CWE195	Signed to Unsigned Conversion Error	conversion-check, signed-overflow-check, unsigned-overflow-check
CWE369	Divide by Zero	div-by-zero-check
CWE401	Memory Leak	memory-leak-check
CWE476	NULL Pointer Dereference	pointer-check
CWE681	Incorrect Conversion Between Numeric Types	conversion-check
CWE690	NULL Deref From Return	pointer-check
CWE761	Free Pointer Not at Start of Buffer	pointer-overflow-check

TABLE 3.2: List of CWEs chosen for RQ2 and combinations of property used to identify them in the ideal scenario in which each CWE is not contamined by other vulnerabilities.

3.5.3 Results

After having computed the performance on the two versions of the dataset for each vulnerability and its predefined set of properties, we have found out that the *precision* of the tool and the overall F1-score decrease respectively by 33% and by 15%, in average. In 13 out of 15 (86,7%) CWEs the precision decreases over 20%. Only in 2 cases out of 15 (13,3%) the precision decreases between 6% and 9%, in average (CWE369, CWE401)

To a deeper analysis, only the performance of CWE369 (Division by Zero) and CWE401 (Memory Leak) decrease poorly in term of *precision*, by 6% and 9% respectively. The vulnerabilities CWE121 (Stack-based Buffer Overflow), CWE124 (Buffer Underwrite), CWE126 (Buffer Overread) and CWE127 (Buffer Underread) are strongly affected by each other (Table 3.4), but the reason is mainly because they share the same properties and are pretty similar in terms of behaviour. CWE690 (Null Dereference from Return), detected by the property *pointer-check*, is the only one for which the precison decreases most, over 60% in average, after having injected new vulnerabilities into the dataset. We can assume that the property generalise too much so to allow the identification of multiple vulnerabilities.

Furthermore, CWE690 has also a very low precision before injecting the dataset, symptom that the property, in this specific case, does not detect the vulnerability well as it is supposed to do. The Common Weakness Enumeraton dictionary, indeed, describe it as *"The product does not check for an error after calling a function that can return with a NULL pointer if the function fails, which leads to a resultant NULL pointer dereference"*[2].

Even more, there is a vulnerability, CWE134 (Use of Externally-Controlled Format String), that is recurrent among vulnerabilities.

Generally speaking, thel tool's precision in detecting specific vulnerabilities, given a predefined set of property supposed to identify them, get worse in presence of multiple vulnerabilities. As explained in Section 3.5.1 if the *precision* of the tool on a given vulnerability type with a set of predefined properties decreases considerably (i.e., the number of false positives on the injected vulnerabilities increase more than true negatives) after having injected further different vulnerable functions, then a software auditor looking for that vulnerability could incur in a wrong detection as the code is affected by related vulnerabilities. Otherwise, if the precision remains similar (lower false positives are found than true negatives), then the properties defined for the specific type of vulnerability (CWE) strongly distinguish between vulnerable code regions related to the CWE. That is, the presence of further vulnerabilities does not affect the detection of the former. Table 3.3 and Figure 3.2 show detailed information about tool performance.

To better understand the relation between properties and vulnerabilities, we have also measured the correlation between CWEs with the purpose of identify (i)

[2]Accessible online at: `https://cwe.mitre.org/data/definitions/690.html` (accessed on September 2018)

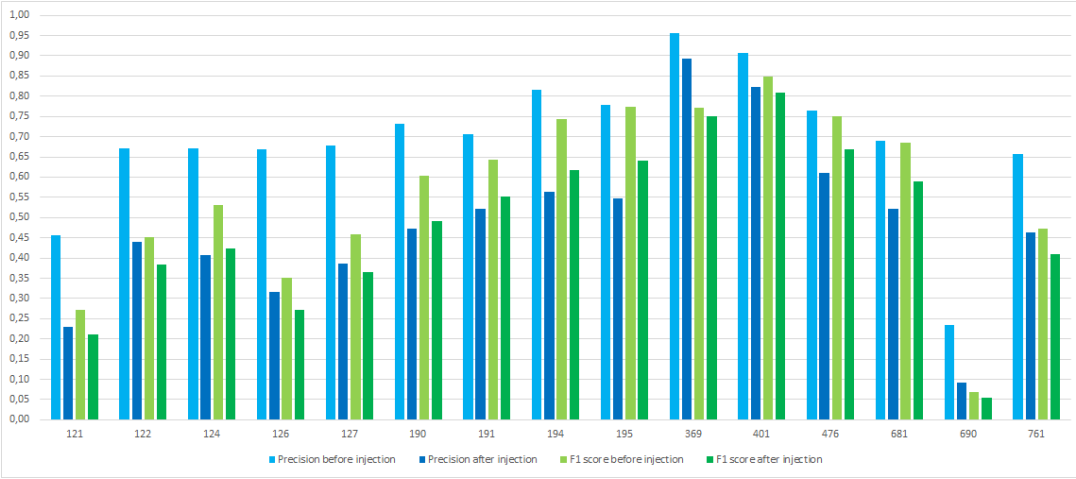

FIGURE 3.2: Tool performance. Each pair of adiacent bars represent the performance (*precision* in blue and *F1-score* in green) before and after having injected the dataset with other vulnerabilities. On the x axes the CWE ID

CWE	Recall	Precision before (FP)	Precision after (FP)
121	0.19	0.46 (21)	0.23 (58)
122	0.34	0.67 (16)	0.44 (41)
124	0.44	0.67 (21)	0.41 (62)
126	0.24	0.67 (11)	0.32 (47)
127	0.35	0.68 (16)	0.39 (54)
190	0.51	0.73 (19)	0.47 (59)
191	0.59	0.71 (25)	0.52 (54)
194	0.68	0.82 (15)	0.56 (50)
195	0.77	0.78 (20)	0.55 (58)
369	0.65	0.96 (3)	0.89 (8)
401	0.80	0.91 (8)	0.82 (17)
476	0.74	0.76 (23)	0.61 (47)
681	0.68	0.69 (15)	0.52 (31)
690	0.04	0.24 (13)	0.09 (40)
761	0.37	0.66 (8)	0.46 (20)

TABLE 3.3: Values of CBMC's recall and precision in average. For the precision (before and after the injection) is reported the number of false positives.

vulnerabilities for which no set of property is know to identify them; and (ii) pairs of vulnerabilities that can affect their respective detection (i.e., they can represent noise of each other if co-occur in source code).

Figures B.1 and B.2 in Appendix B show the correlation between vulnerabilities present in the dataset. Figure 3.3 shows the most significant correlations having *p-value* $< .05$ and correlation coefficient ≥ 0.70. Among them we have found a recurrent vulnerability type, CWE134 (Use of Externally-Controlled Format String) with high correlation with CWE124 (0.73), CWE191 (0.72), CWE195 (0.71) and CWE78 (0.78). Others important correlations occur between CWE191 (Integer Underflow) and CWE194 (0.78), CWE195 (0.72), CWE78 (0.71). The last one refers to *Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)*.

Table 3.4 shows the vulnerabilities and the number of false positives that disturb the detection of each of the considered CWEs in the injection scenario. All these vulnerabilities could affect the detection of the correlated ones and a software auditors should keep this possibility in mind during software auditing activities.

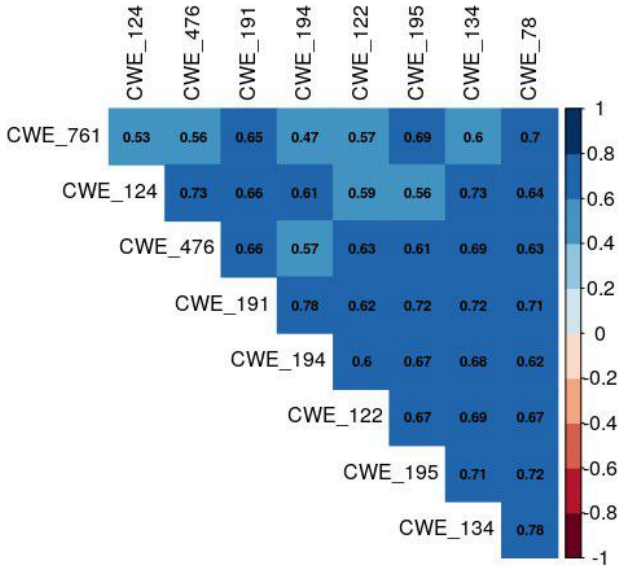

FIGURE 3.3: Significant correlations with an *p-value* < *0.05* and correlation coefficient greater than 0.7

Summary for RQ2. The presence of multiple vulnerabilities in the source code can affect the detection of correlated vulnerabilities. Only two properties (div-by-zero-check, memory-leak-check), from those considered, have been seen to discriminate well among several vulnerabilities.

CWE	121	122	124	126	127	190	191	194	195	369	401	476	681	690	761
78	**3/4**	**2/7**	**2/4**	**6/6**	**2/5**	**5/6**	**3/5**	**3/3**	**3/7**	**0/5**	**0/4**	**2/4**	**1/2**	**2/2**	**5/7**
114	**0/1**	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0
121	0/0	**1/17**	**11/15**	**5/15**	**3/16**	**3/26**	**1/17**	**2/11**	**4/13**	**0/15**	**0/13**	**3/14**	**1/7**	**3/9**	**1/13**
122	**3/8**	0/0	**2/4**	**2/6**	**5/8**	**1/6**	**3/7**	**1/7**	**2/6**	**1/12**	**1/9**	**1/7**	**1/3**	**3/4**	**0/5**
123	0/0	**1/1**	0/0	0/0	0/0	0/0	0/0	0/0	**1/1**	0/0	**0/1**	0/0	0/0	0/0	**0/1**
124	**0/4**	**1/3**	0/0	**2/4**	**2/6**	**0/7**	**0/5**	**0/6**	**0/4**	**0/6**	**2/5**	**2/3**	**1/3**	**0/1**	**0/5**
126	**0/1**	**0/2**	**0/1**	0/0	**1/3**	**1/5**	**0/1**	**0/2**	**2/2**	**0/1**	**0/5**	**0/1**	**1/2**	**0/3**	**0/3**
127	**2/5**	**1/3**	**1/3**	0/0	**0/2**	**1/6**	**1/4**	**0/6**	**0/7**	**0/6**	**0/5**	**0/7**	**1/2**	**0/2**	**0/4**
134	**7/8**	**3/5**	**6/8**	**5/8**	**6/8**	**6/10**	**4/8**	**4/6**	**5/8**	**0/5**	**0/2**	**5/9**	**0/4**	**4/8**	**1/1**
190	**0/9**	**0/8**	**2/17**	**2/9**	**1/10**	**3/6**	**10/13**	**9/16**	**4/9**	**0/11**	**0/14**	**1/7**	**2/6**	**0/13**	**0/13**
191	**2/12**	**0/9**	**1/12**	**0/9**	**3/12**	**6/10**	0/0	**3/6**	**6/10**	**0/6**	**0/5**	**1/9**	**2/3**	**2/12**	**0/6**
194	**1/4**	**0/1**	**2/3**	**2/3**	**2/2**	0/0	**0/2**	0/0	**4/4**	**0/4**	**0/4**	**1/4**	**1/2**	**3/5**	**0/5**
195	**1/1**	**3/3**	**0/2**	**0/3**	**1/1**	**2/4**	**1/2**	**1/1**	0/0	**0/2**	**0/1**	**1/4**	0/0	**1/1**	**0/2**
197	**2/2**	**1/4**	0/0	**2/3**	**1/1**	**0/4**	**0/1**	**2/2**	**0/2**	**0/2**	**0/2**	0/0	**1/1**	0/0	**0/1**
252	**1/3**	**0/2**	**1/2**	**0/3**	**2/2**	**0/4**	**0/3**	**0/2**	**0/1**	**0/1**	0/0	**1/2**	0/0	**0/3**	**0/2**
253	**0/1**	**0/3**	**1/1**	**2/3**	**0/1**	0/0	**0/3**	**0/2**	0/0	0/0	**0/2**	**0/1**	**0/1**	**1/3**	**0/1**
364	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	**0/1**
366	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**
369	**3/3**	**3/5**	**1/1**	**1/2**	**1/2**	**0/3**	**0/1**	**2/3**	**0/2**	0/0	**0/2**	**1/3**	0/0	**1/1**	**0/4**
377	0/0	**1/1**	0/0	0/0	**3/3**	0/0	0/0	0/0	**2/2**	0/0	0/0	0/0	0/0	0/0	0/0
390	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0
391	0/0	0/0	0/0	0/0	**0/1**	**0/1**	0/0	0/0	0/0	**1/1**	0/0	0/0	0/0	**0/1**	**0/1**
398	**0/1**	0/0	0/0	0/0	0/0	**0/3**	**0/1**	**0/2**	**0/1**	0/0	**0/1**	0/0	0/0	0/0	**0/1**
400	**4/4**	0/0	**1/2**	0/0	0/0	**1/3**	**1/2**	**2/4**	**0/1**	**0/2**	**0/2**	**1/3**	0/0	**0/2**	**0/2**
401	**0/2**	**0/3**	**0/2**	**0/4**	**0/2**	**0/7**	**0/4**	**0/3**	**0/2**	**0/2**	0/0	**0/1**	0/0	**0/4**	**0/4**
404	**1/1**	0/0	0/0	0/0	**0/1**	0/0	0/0	**1/3**	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0
415	**0/0**	0/0	**2/3**	0/0	0/0	**1/1**	0/0	0/0	0/0	0/0	**2/2**	**1/2**	0/0	**1/3**	0/0
416	**1/1**	0/0	0/0	0/0	0/0	**0/1**	**0/1**	0/0	0/0	**0/1**	**0/1**	0/0	**0/1**	**0/1**	**0/1**
426	**0/1**	0/0	0/0	**0/2**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
427	**2/2**	**1/2**	**1/2**	**0/1**	0/0	**2/2**	**3/4**	**2/2**	**0/2**	**0/2**	0/0	**0/1**	**1/1**	0/0	**0/1**
457	**0/2**	0/0	**0/2**	**0/2**	**0/2**	**0/3**	0/0	0/0	**0/2**	**0/3**	**0/2**	**0/5**	**0/2**	**0/1**	**0/2**
459	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0
464	**0/2**	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
467	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0
468	**0/1**	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
469	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
476	0/0	**1/1**	**1/1**	**0/1**	0/0	**0/3**	0/0	**0/1**	0/0	**0/2**	0/0	0/0	0/0	0/0	**0/2**
479	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0
480	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	**0/2**	0/0	0/0	**0/1**	0/0
510	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**1/1**	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0
511	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0
526	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
546	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	**0/2**	0/0	0/0	0/0	0/0	**0/1**	**0/1**
561	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
563	**0/2**	0/0	0/0	**0/3**	**0/1**	**0/3**	**0/1**	0/0	**0/1**	0/0	**0/1**	**0/2**	0/0	**0/3**	0/0
570	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0
587	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
588	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0
590	**1/3**	**0/2**	**1/3**	0/0	**3/4**	**3/8**	**1/3**	**1/2**	0/0	**1/2**	**1/1**	**0/1**	0/0	**0/4**	**1/1**
606	**2/2**	**1/2**	**0/1**	**1/1**	**1/3**	0/0	**0/2**	**0/1**	**1/1**	**0/1**	**0/3**	0/0	**0/2**	0/0	**1/2**
617	0/0	0/0	**1/1**	0/0	0/0	0/0	**1/1**	**1/2**	**2/2**	0/0	0/0	0/0	0/0	**1/1**	**1/1**
665	**0/1**	0/0	**0/1**	**0/2**	**0/1**	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
666	**0/1**	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/3**	0/0
675	0/0	**2/3**	0/0	0/0	0/0	**2/2**	0/0	0/0	**1/1**	**2/3**	0/0	0/0	0/0	0/0	**1/1**
680	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	**1/1**	0/0	0/0
681	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**2/2**	0/0
685	0/0	**0/1**	**0/1**	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0
688	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0
690	**0/3**	**0/5**	**1/3**	**1/2**	**0/2**	**0/3**	**0/2**	**0/2**	**0/1**	**0/2**	**2/3**	**1/4**	**1/3**	0/0	**0/1**
758	**0/1**	**1/1**	**0/1**	**1/2**	0/0	**0/1**	0/0	0/0	0/0	0/0	**0/1**	**0/1**	0/0	**2/4**	**0/3**
761	**0/1**	**1/3**	**0/2**	**0/2**	0/0	**0/5**	**0/3**	**0/1**	**0/2**	0/0	**0/1**	**1/2**	**0/1**	**0/1**	0/0
773	0/0	**0/1**	**1/2**	0/0	0/0	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0
775	0/0	0/0	0/0	0/0	0/0	**0/1**	**0/1**	0/0	0/0	0/0	**1/1**	0/0	0/0	0/0	0/0
789	0/0	0/0	0/0	**2/2**	0/0	**1/1**	**0/2**	**0/1**	0/0	0/0	**0/1**	0/0	**0/2**	0/0	**0/1**
832	0/0	0/0	0/0	0/0	**0/1**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

TABLE 3.4: Number of *false positives* of each injected CWEs (rows) for each analysed CWE (column). The table should be read top-down, left-right. The numerator indicates the number of false positives, while the denominator indicates the total number of functions present in the dataset for the considered CWE. A denominator equal to 0 means that that CWE in the row was not present in the injected dataset for the CWE in the column.

3.6 Loop Unwinds Optimal Values

RQ3. *Which are optimal values of loop unwinds for which precision and recall are maximized?*

3.6.1 Design

In BMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using an efficient SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. If the formula is not satisfiable, the program can be unwound more to determine if a longer counterexample exists. CBMC can formally verify such bounds by means of unwinding assertions. Once this bound is established, it can prove the absence of errors. The basic idea of CBMC is to model the computation of the programs up to a depth k. Technically, this is achieved by a process that essentially amounts to unwinding loops. Even though CBMC can deal with unwinding in an automatic fashion to determine an upper bound on the number of loop iterations, it also provides an option, named `unwind nr`, that models the computation of the programs up to a particular depth defined by the user, i.e. unwind the loops up to `nr` iterations.

To answer our last research question we have run the tool with several unwind values of k - from 1 to 2000 incremented by 100 - for each function and given the tool a reasonable timeout of ten minutes to end the computation. For each value of k we have compared the results in terms of F1-score and number of completed verifications within the timebound.

Since the question is about understanding the role of k in the detection of vulnerable code regions in a setting where multiple vulnerabilities can co-occur, we have used the same dataset for RQ2 as defined in Section 3.5.2

3.6.2 Results

Given a set of dimensional points, a *skyline* point should have the property on each dimension no worse than any other points. Given a set of d dimensional points P we say that one point p1 dominates another point p2 if and only if:

1. p1 is better than or equal to p2 on all dimensions, and
2. p1 is better than p2 on at least one dimension.

Here "better" can be either "smaller better" (minimum skyline) or "larger better" (maximum skyline) or any other criteria measurable and comparable. The skyline points will be the ones which will not be dominated by any other points in P.

Figure 3.4 reports the skylines for the unwind values for which precision and recall are maximised. Values between 1 and 1000 seem to give the better overall results in terms of both precision and recall.

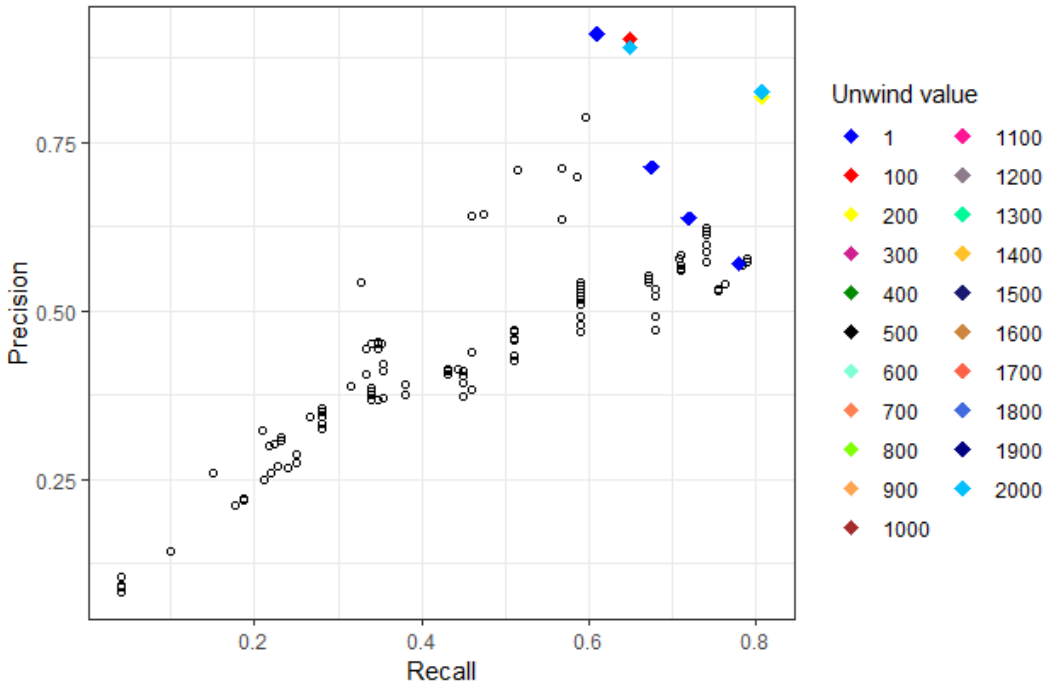

FIGURE 3.4: Plot precision and recall. The Skylines points maximizing both dimensions are bold.

Figures 3.5 and 3.6 show the average F1-score and number of completed verifications within ten minutes in function of k, respectively.

Summary for RQ3. There exist several local optima for which precision and recall are maximized. However, small values for unwind, e.g. 1 and 200, are *sufficient* to maximize both dimensions. This is an importan result, as if k increases, the number of verifications that terminates within the timebound decreases considerably.

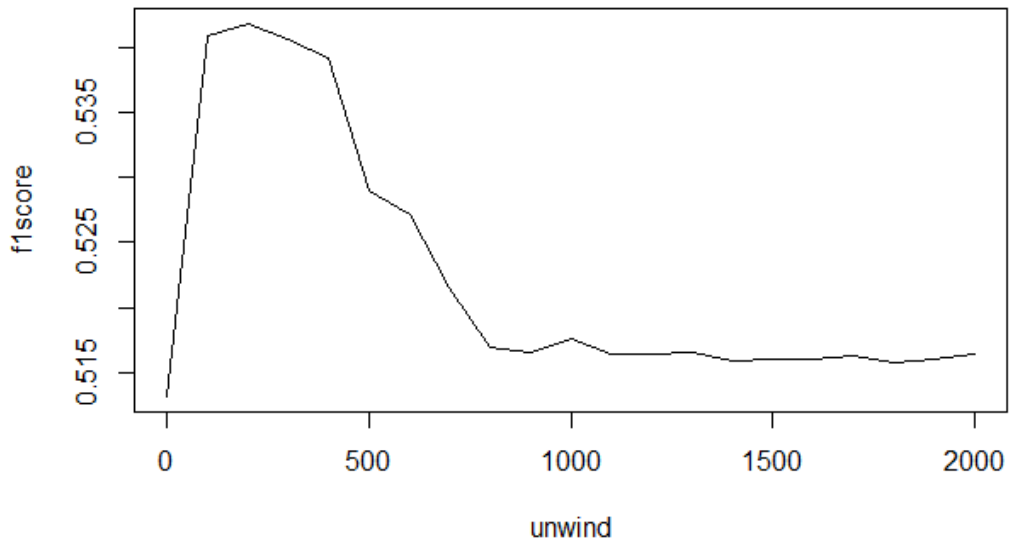

FIGURE 3.5: Average F1-score in function of k

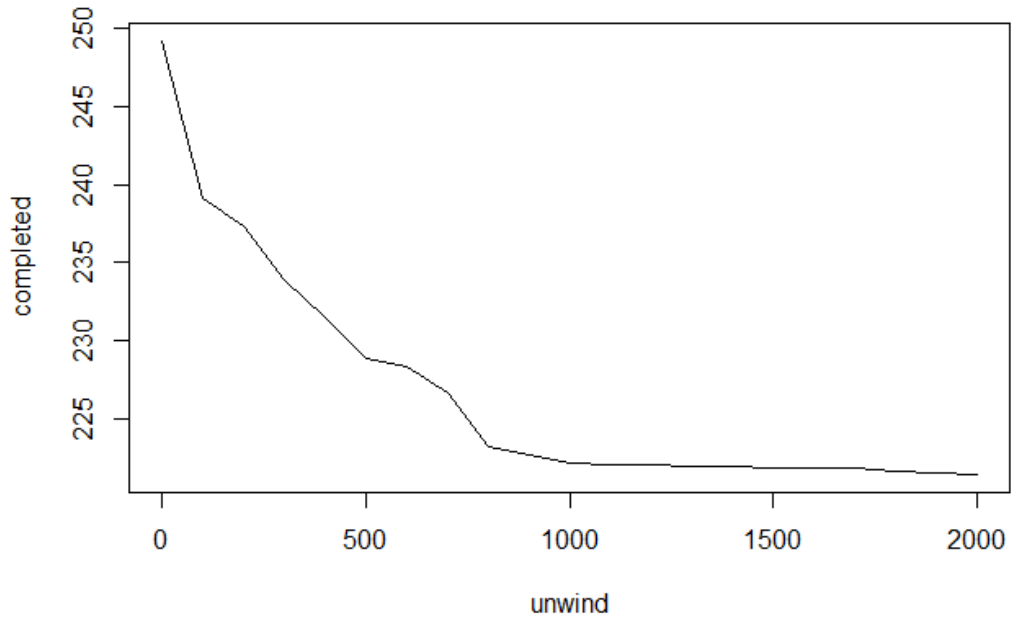

FIGURE 3.6: Average number of completed verifications (within ten minutes) in function of k

3.7 Threats to Validity

3.7.1 Threats to Construct and Internal Validity

The Juliet Test Suite that we used is comprised of computer-generated code, which may not have the same characteristics as human-generated code (e.g. it is more isolated than human-generated code and less complex). Thus, further research is necessary to equally examine the weaknesses in human-generated code. We tested several common weaknesses in a systematic manner, but in order to show the true coverage of Bounded Model Checking technique in security context, representative complex human-made code containing multiple CWEs should be tested.

Furthermore one might argue that the noise in the detection of the vulnerabilities is given by the wrong choice of properties for each CWE. However, we opted for a conservative choice of these properties, including no more than three properties for CWE. In most cases, the chosen CWE needed only one property to be correctly detected.

3.7.2 Threats to External Validity

Threats to external validity affect the generalisation of our results. We carried out experiments on a total of 5,355 (1,005 for RQ1 and 4,350 for RQ2) test cases uniformly taken from 67 different CWEs belonging to the Juliet Test Suite composed of more than 64,000 test cases over 118 CWEs with different characteristics. Since the dataset was composed of computer-generated code created on purpose to represent each CWE, we are aware that our findings may and may not be directly applicable to more complex environments, such as industry, due to the complexity of their code. However the replication of our study on open-source and industrial projects is part of our future research agenda.

Chapter 4

Conclusion and Future Directions

A sad truth is that in most projects verification (of any kind) is pushed to the very end of the development cycle. It is highly recommendable to use formal verification techniques as early as possible in the development cycle, above all for safety critical systems. The correct implementation of such techniques determines the correctness of the system relying on them. Therefore, strong attention should be posed on them. Below we discuss future directions of our work.

4.1 Replication study with SatAbs

CBMC and SatAbs are tools able to check whether a property is always valid for any execution of the program. They verify array bounds, pointer safety, exceptions and user-specified assertions. Both tools are able to reason about a class of bugs that has so far gone annoticed by many verification tools. However, the algorithms implemented by CBMC and SatAbs are complementary, and often, one tool is able to solve a problem that the other cannot solve. Furthermore, comparing static analysis tool results on natural code is complicated because different tools report results in different manners. However, CBMC and SatAbs provide a similar result format, making the comparison easier to perform.

Thus, we want to analyse their behaviour and verify whether their combined use could be beneficial in security context to facilitate software auditors in the localization and the repair of the vulnerabilities. In our research agenda we plan to perform a replication study on SatAbs and compare the results with the current work.

4.2 User Study

Can CBMC (really) help in identifying vulnerabilities? (Oliveira et al., 2014) showed that developers can have difficulties correlating a particular learned vulnerability or security information with their current working task, and priming or explicitly cueing about vulnerabilities on-the-spot is a powerful mechanism to make developers aware about potential vulnerabilities. In future works we aim at conducting a qualitative study involving developers to study whether these tools can be used as support to make them aware about potential vulnerabilities in their code.

Appendix A

Software Errors

Software Testing is important because software bugs could be expensive or even dangerous. Software bugs can potentially cause monetary and human loss, history is full of such examples.

- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving three people dead and critically injuring three others.
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocent lives.
- In May of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.
- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history.
- In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
- In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.
- Nissan cars had to recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accidents due to this software failure.
- Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one-point store served coffee for free as they unable to process the transaction.
- Some of the Amazon's third-party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.
- Vulnerability in Window 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.

Appendix B

CWEs

The Common Weakness Enumeration Specification (CWE) provides a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities as they are found in code, design, or system architecture. Each individual CWE represents a single vulnerability type. CWE is currently maintained by the MITRE Corporation with support from the National Cyber Security Division (DHS). A detailed CWE list is currently available at the MITRE website; this list provides a detailed definition for each individual CWE Targeted to developers and security practitioners, the Common Weakness Enumeration (CWE) is a formal list of software weakness types created to:

- Serve as a common language for describing software security weaknesses in architecture, design, or code.
- Serve as a standard measuring stick for software security tools targeting these weaknesses.
- Provide a common baseline standard for weakness identification, mitigation, and prevention efforts.

Below are listed the analysed CWEs contained in the Juliet Test suite as described in Chapter 3.

CWE ID	Name	No Functions
CWE121	Stack-based Buffer Overflow	7889
CWE122	Heap-based Buffer Overflow	9652
CWE123	Write What Where Condition	168
CWE124	Buffer Underwrite	3319
CWE126	Buffer Overread	2297
CWE127	Buffer Underread	3319
CWE134	Uncontrolled Format String	4929
CWE188	Reliance on Data Memory Layout	38
CWE190	Integer Overflow and Wraparound	6458
CWE191	Integer Underflow	4732
CWE194	Unexpected Sign Extension	1876
CWE195	Signed to Unsigned Conversion Error	1876
CWE196	Unsigned to Signed Conversion Error	18
CWE197	Numeric Truncation Error	1418
CWE242	Use of Inherently Dangerous Function	18
CWE364	Signal Handler Race Condition	18
CWE366	Race Condition Within Thread	36
CWE367	TOC TOU	36
CWE369	Divide by Zero	1480
CWE390	Error Without Action	72
CWE391	Unchecked Error Condition	54
CWE398	Poor Code Quality	181
CWE400	Resource Exhaustion	1234
CWE401	Memory Leak	2710
CWE415	Double Free	1648
CWE416	Use After Free	150
CWE426	Untrusted Search Path	224
CWE427	Uncontrolled Search Path Element	560
CWE457	Use of Uninitialized Variable	1083
CWE459	Incomplete Cleanup	36
CWE464	Addition of Data Structure Sentinel	56
CWE467	Use of sizeof on Pointer Type	54
CWE468	Incorrect Pointer Scaling	36
CWE469	Use of Pointer Subtraction to Determine Size	36
CWE475	Undefined Behavior for Input to API	36
CWE476	NULL Pointer Dereference	372
CWE478	Missing Default Case in Switch	18
CWE479	Signal Handler Use of Non Reentrant Function	18
CWE480	Use of Incorrect Operator	18
CWE481	Assigning Instead of Comparing	18
CWE482	Comparing Instead of Assigning	18
CWE483	Incorrect Block Delimitation	20
CWE484	Omitted Break Statement in Switch	18
CWE510	Trapdoor	70
CWE526	Info Exposure Environment Variables	18
CWE546	Suspicious Comment	90
CWE563	Unused Variable	366
CWE570	Expression Always False	16
CWE571	Expression Always True	16
CWE587	Assignment of Fixed Address to Pointer	18
CWE588	Attempt to Access Child of Non Structure Pointer	50
CWE590	Free Memory Not on Heap	4231
CWE591	Sensitive Data Storage in Improperly Locked Memory	112
CWE605	Multiple Binds Same Port	18
CWE606	Unchecked Loop Condition	560
CWE617	Reachable Assertion	354
CWE665	Improper Initialization	224
CWE666	Operation on Resource in Wrong Phase of Lifetime	90
CWE667	Improper Locking	18
CWE681	Incorrect Conversion Between Numeric Types	54
CWE685	Function Call With Incorrect Number of Arguments	18
CWE688	Function Call With Incorrect Variable or Reference as Argument	18
CWE690	NULL Deref From Return	1564
CWE758	Undefined Behavior	365
CWE761	Free Pointer Not at Start of Buffer	762
CWE789	Uncontrolled Mem Alloc	1644
CWE832	Unlock of Resource That is Not Locked	18
CWE843	Type Confusion	100

TABLE B.1: List of the 67 CWEs chosen for RQ1 in Section 3.4.

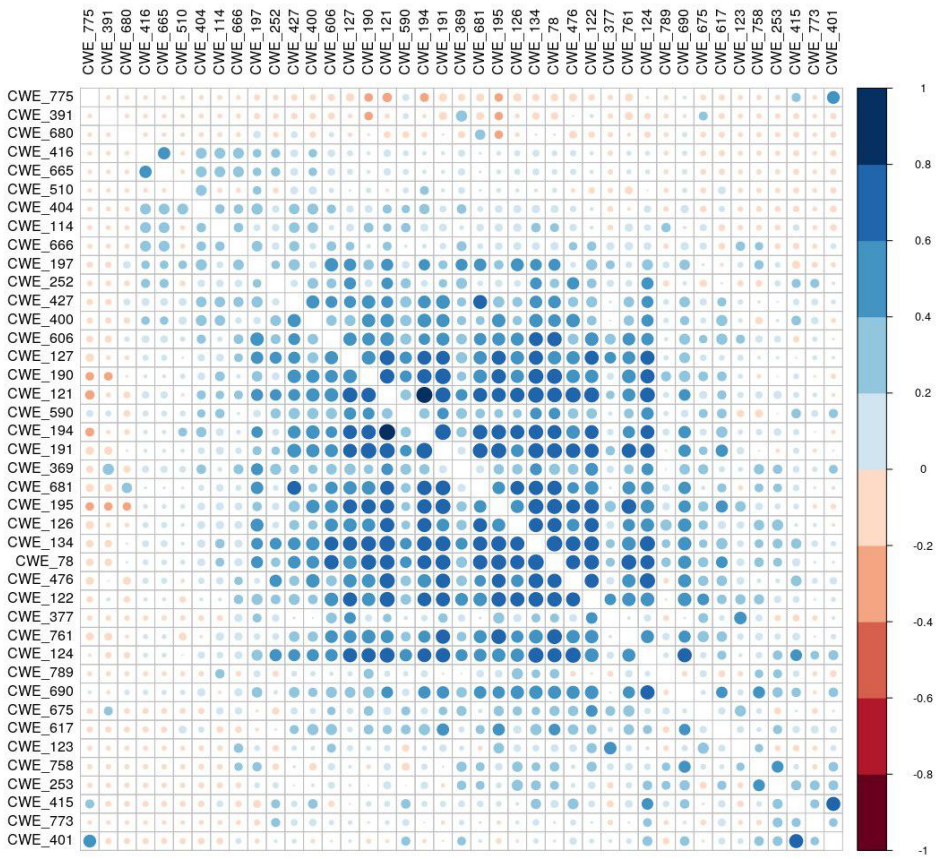

FIGURE B.1: Correlation matrix among the analysed and injected cwes

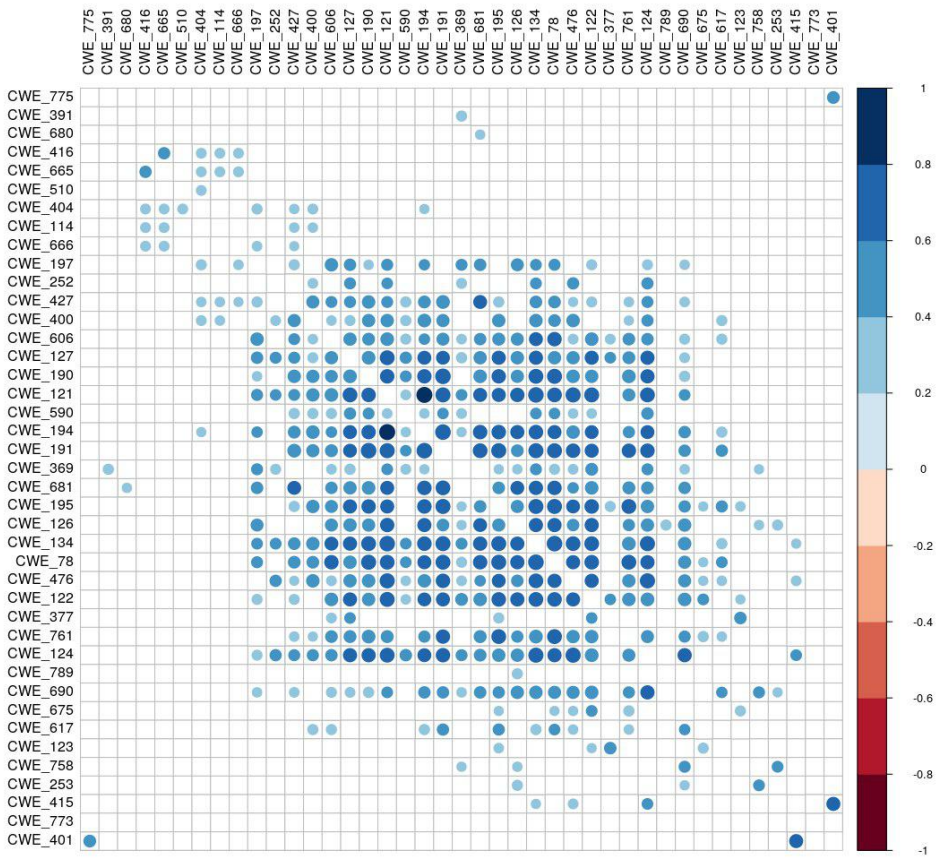

FIGURE B.2: Insignificant values have been filtered out from the correlation matrix

Bibliography

Agten, Pieter, Bart Jacobs, and Frank Piessens (Jan. 2015). "Sound Modular Verification of C Code Executing in an Unverified Context". In: *SIGPLAN Not.* 50.1, pp. 581–594. ISSN: 0362-1340. DOI: 10.1145/2775051.2676972. URL: http://doi.acm.org/10.1145/2775051.2676972.

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press. ISBN: 026202649X, 9780262026499.

Ball, Thomas and Sriram K. Rajamani (Jan. 2002). "The SLAM Project: Debugging System Software via Static Analysis". In: *SIGPLAN Not.* 37.1, pp. 1–3. ISSN: 0362-1340. DOI: 10.1145/565816.503274. URL: http://doi.acm.org/10.1145/565816.503274.

Biere, Armin et al. (1999). "Symbolic model checking without BDDs." In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer, pp. 193–207.

Biere, Armin et al. (2003). "Bounded model checking." In: *Advances in computers* 58.11, pp. 117–148.

Clang Static Analyzers. http://clang-analyzer.llvm.org/. Accessed: June, 2018.

Clarke, Edmund, Daniel Kroening, and Flavio Lerda (2004). "A tool for checking ANSI-C programs." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 168–176.

Clarke, Edmund et al. (2005). "SATABS: SAT-based predicate abstraction for ANSI-C". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, pp. 570–574.

Clarke, Edmund M. and E. Allen Emerson (1982). "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logic of Programs, Workshop*. Berlin, Heidelberg: Springer-Verlag, pp. 52–71. ISBN: 3-540-11212-X. URL: http://dl.acm.org/citation.cfm?id=648063.747438.

Dahlweid, Markus et al. (2009). "VCC: Contract-based modular verification of concurrent C". In: *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, pp. 429–430.

Dijkstra, E.W. (1969). "Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee." In: p. 16.

Dunn, Olive Jean (1964). "Multiple comparisons using rank sums". In: *Technometrics* 6.3, pp. 241–252.

Eisner, Cindy (2005). "Formal verification of software source code through semi-automatic modeling". In: *Software & Systems Modeling* 4.1, pp. 14–31.

Filliâtre, Jean-Christophe and Claude Marché (2004). "Multi-prover verification of C programs". In: *International Conference on Formal Engineering Methods*. Springer, pp. 15–29.

Flawfinder. http://www.dwheeler.com/flawfinder/. Accessed: June, 2018.

Graf, Susanne and Hassen Saïdi (1997). "Construction of Abstract State Graphs with PVS". In: *Proceedings of the 9th International Conference on Computer Aided Verification*. CAV '97. London, UK, UK: Springer-Verlag, pp. 72–83. ISBN: 3-540-63166-6. URL: http://dl.acm.org/citation.cfm?id=647766.733618.

Holzmann, Gerard J (2000). "Logic verification of ANSI-C code with SPIN". In: *International SPIN Workshop on Model Checking of Software*. Springer, pp. 131–147.

IEEE (1997). "1028-1997 - IEEE Standard for Software Reviews". In: *clause 3.2*.

Nethercote, Nicholas and Julian Seward (June 2007). "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6, pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: http://doi.acm.org/10.1145/1273442.1250746.

Oliveira, Daniela et al. (2014). "It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACSAC '14. New Orleans, Louisiana, USA: ACM, pp. 296–305. ISBN: 978-1-4503-3005-3. DOI: 10.1145/2664243.2664254. URL: http://doi.acm.org/10.1145/2664243.2664254.

Perl, Henning et al. (2015). "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, pp. 426–437. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813604. URL: http://doi.acm.org/10.1145/2810103.2813604.