Detection of Security Vulnerabilities Induced by Integer Errors

Salim Yahia Kissi[1][a], Yassamine Seladji[1][b] and Rabéa Ameur-Boulifa[2][c]
[1]*LRIT, University of Abou Bekr Belkaid, Tlemcen, Algeria*
[2]*LTCI, Télécom Paris, Institut Polytechnique de Paris, France*

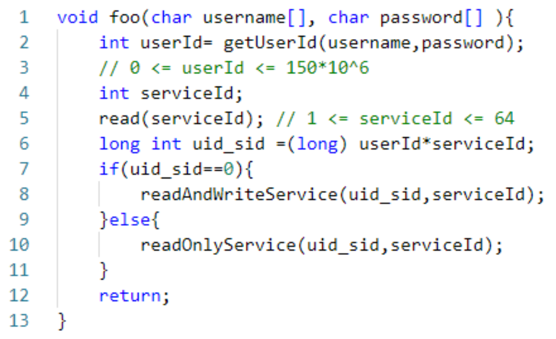
Keywords: Security Vulnerability, Memory Errors, Software Analysis, Satisfiability Analysis, Integer Overflow.

Abstract: Sometimes computing platforms, e.g. storage device, compilers, operating systems used to execute software programs make them misbehave, this type of issues could be exploited by attackers to access sensitive data and compromise the system. This paper presents an automatable approach for detecting such security vulnerabilities due to improper execution environment. Specifically, the advocated approach targets the detection of security vulnerabilities in the software caused by memory overflows such as integer overflow. Based on analysis of the source code and by using a knowledge base gathering common execution platform issues and known restrictions, the paper proposes a framework able to infer the required assertions, without manual code annotations and rewriting, for generating logical formulas that can be used to reveal potential code weaknesses.

1 INTRODUCTION

With the aim of classifying the software weaknesses (types of vulnerabilities) organizations define a huge number of different data-sources to be used by software developers to avoid particular attacks. Data-sources are generally descriptions and set of characteristics, which require extra effort to interpret, assess and demonstrate potential risks. In practice, in companies this kind of tasks is being carried out through review processes and conduct audit sessions comprising a wide range of experiments. However, manual reviews can be time-consuming and costly to the companies in terms of resources, and they can fall short in detecting security vulnerabilities. From a software engineering perspective, finding out security vulnerabilities is not a trivial task for several reasons including software weaknesses are often written in an informal style, they use various technical information (concepts), and they require domain expertise to interpret them.

In this work we consider a well known threat to systems security: integers errors. These errors result from integer operations, including arithmetic overflow, oversized shift, division-by-zero, lossy truncation and sign misinterpretation, that can be manipulated by malicious users. Our focus here is on arithmetic overflow within the C standard. One reason why such errors remain serious source of problems is that it is difficult for programmers to reason about integers semantics (Dietz et al., 2015). Importantly, there are unintentional numerical bugs that are caused by the execution environment, including the widely-used applications and libraries, but they can also be caused by the features of the execution platforms (e.g. compilers, operating systems). This is not merely a fringe case, but it is observable already on small programs. To illustrate the problem, consider the code snippet shown in Figure 1. The code shows a method



```
void foo(char username[], char password[] ){
    int userId= getUserId(username,password);
    // 0 <= userId <= 150*10^6
    int serviceId;
    read(serviceId); // 1 <= serviceId <= 64
    long int uid_sid =(long) userId*serviceId;
    if(uid_sid==0){
        readAndWriteService(uid_sid,serviceId);
    }else{
        readOnlyService(uid_sid,serviceId);
    }
    return;
}
```

Figure 1: Access Control Sample Code.

which computes the access rights of an user to services from his credentials (username and password) provided as inputs. It specifies that read and write access rights to services are granted only to a spe-

[a] https://orcid.org/0000-0002-9222-0291
[b] https://orcid.org/0000-0003-2778-7555
[c] https://orcid.org/0000-0002-2471-8012

Kissi, S., Seladji, Y. and Ameur-Boulifa, R.
Detection of Security Vulnerabilities Induced by Integer Errors.
DOI: 10.5220/0010551301770184
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 177-184
ISBN: 978-989-758-523-4

cific user the administrator (its *userId* has a value of 0) and other users (their *userId* are different from 0) should access with reading rights only. Note that the access right is obtained by multiplying the user identity and the service identity (line 6). Compiling and running this code with gcc on Windows operating system 32/64 bits using 64 bits CPU (x86_64 such Intel and AMD processors) produces a program that grants read and write permissions to an unauthorized user. This security problem stems from the arithmetic operation, the multiplication overflows resulting in undefined behavior.

Detection of computer security vulnerabilities that are generated inadvertently by runtime platforms is still an open problem (Hohnka et al., 2019). As a number of unknown (numerical) bugs in widely used open source software packages (and even in safe integer libraries) inadvertently create vulnerabilities in the resulting code. This work presents an approach to formally detect misbehaviour that can lead to security concerns. We are able to detect exploits of C programs induced by an unintentional arithmetic overflow caused by the execution platform. To allow the analysis of programs by integrating execution platform, we suggest enhancing symbolic execution models with characteristics of computer system, and to offer, from the same model, software-specification and in addition, hardware-specification analyses. The model gives a fully formal and analyzable semantics for C code in terms of a logical formula. And the use of SMT-solvers allows to decide if it is satisfiable.

Contribution. This article presents our approach that provides the means to formally specify the software weaknesses, and to evaluate their potential technical impact using formal proofs. The approach based on static analysis is designed to evaluate the security impact of integer errors, in particular integer overflows by analysing memory error exploitations over programs. Formally speaking, we use symbolic execution to generate program constraint (PC), and get security constraint (SC) from predefined security requirements. In addition, based on a precise knowledge on the execution context of the analysed program (EC), we propose to solve the statement: $EC \vdash PC \wedge \neg SC$ we seek to find out if there is an assignment of values to program inputs – executed in a certain context – which could satisfy PC but violates SC.

Although we focus on the integer errors and programs in C language, we believe that our approach is quite general. It can be applied to analyze other sources of problems (e.g. software/hardware exceptions, pointer aliasing) but also other programming languages.

The paper flow is as follows: Section 2 presents a high level view of memory error, in particular buffer overflow and integer errors. This section is then followed by an overview of the proposed end-to-end approach for the specification and verification of their potential security impact (Section 3). In Section 4, we present existing approaches that dealt with security vulnerabilities detection. Section 5 concludes the paper and discusses possible directions of this work.

2 MEMORY ERRORS

Memory errors in C and C++ programs are probably the best-known software vulnerabilities (Van der Veen et al., 2012). These errors include buffer overflows, use of pointer references, format string vulnerabilities and arithmetic vulnerabilities. This paper focuses on a subclass of software vulnerabilities: buffer overflow and integer errors, which address mainly the memory corruption errors.

Buffer Overflow. Buffers are areas of memory expect to hold data. When a variable is declared in a program, space is reserved for it and memory is dynamically allocated at run-time. A buffer overflow may occur when size of data is larger than the buffer size. Then while writing data into a buffer, the program overruns the buffer's boundary and overwrites adjacent memory space, which results in unpredictable program behaviour, including crashes or incorrect results.

Integer Errors. Most typed programming languages have fixed memory size for simple data type that fits with the word size of the underlying machine. Two kinds of integer errors that can lead to exploitable vulnerabilities exist: arithmetic overflows and sign conversion errors. The first occurs when the result of an arithmetic operation is a numeric value that is greater in magnitude than its storage location. While the second occurs when the programmer defines an integer, it is assumed to be a signed integer but it is converted to an unsigned integer.

Basically, this kind of memory-safety issues can yield result far than expected when the impacted memory space is accessed. Even worse, they could become security exploits (security vulnerabilities) (Younan et al., 2004) if the result of the memory access is used to perform unauthorized actions and gain unauthorized access to privileged areas. Erroneous programs can allow a malicious actor to run code, install malware, and steal, destroy or modify sensitive data.

Nowadays there are more and more software companies, organizations and developers such as Source Code Analysis Laboratory (SCALe) (Seacord et al., 2012) and CERN computer security (CER,) sharing good and bad programming practices to develop higher quality software and avoid bugs. These practices are generic coding conventions and recommendations which may apply (and not apply) for software written with a particular programming language. However, such coding standards are presented in an informal style, and are not located in one single place. Our work proposes a proof-based approach that takes advantages of these coding conventions and developers knowledge to discover potential vulnerabilities that can be hidden in a code. We transform some safety-related practices that can lead to security exploits from their informal specification to exploitable safety-properties that can be automatically verified over a program. For example, integer overflow is positioned in the ”Top 25 Most Dangerous Software Errors” (CWE,). On hardware platforms, the range of two's complement representation of an n-bit signed integer is $-2^{n-1} \dots 2^{n-1}-1$, which is represented in the computer as depicted in Figure 2.

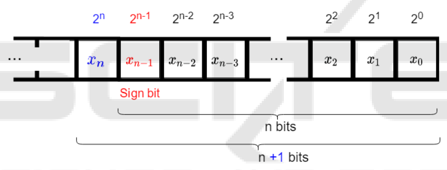


Figure 2: Signed Magnitude Representation.

where $x_i \in \{0,1\}$. If the most significant digit x_{n-1} is a 0, the number is evaluated as an unsigned (positive) integer, otherwise the number is a negative integer. The value of the integer can then be calculated by the following formula:

$$-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

Detecting integer overflows is non-trivial because overflow behaviours are not always bugs. In particular, some International Standard like C99 imposes no requirements, the result of evaluating an integer overflow in C implementation is an undefined behaviour.

3 APPROACH

We propose an approach that enables the detection of potential security vulnerability and the formalization of security weaknesses. Our approach relies on formal proofs for the detection of security exploits caused by intentional or unintentional safety bugs. By taking an interest in the knowledge about undefined behaviour in programs that can result in potential security exploits, we focus mainly on verifying whether those security exploits can occur or not. Figure 3 illustrates our approach and highlights the relevant phases for the verification of security exploits over the program to analyze. We aim at separating the duties and make the distinction between the main actors in our approach; the specification expert(s) and the developer(s). In a preliminary phase, the specification expert(s) carries out the extraction of logical formulas from software vulnerabilities directories. This task consists of translating undefined behaviour into exploitable formulas. It results in a set of generic logical formulas saved in a database (we refer to as Safety Knowledge Base) that could be used in different analyses.

3.1 Safety Knowledge Base

The key idea of our approach is to build a Safety Knowledge Base as a centralized repository for gathering formal specifications, which are logical formulas so-called safety-properties. The database will be created from errors and recommendations reported in software (vulnerabilities) directories. Regarding integer errors as reported in CERT directory ”INT32-C. Ensure that operations on signed integers do not result in overflow” [1], the developers have listed 15 possible operations among 36 that could lead to overflow. Based on the reported result we can distinguish several patterns binary operations and unary operations on different data types: *signed char, short, int, long, long long*, and also type of stdint library. For each operation with a given type, in the style of what has been done in Frama-C (Kosmatov and Signoles, 2013).

The advocated approach relies on using logical formula (assertions) to uncover security vulnerability due to overflow errors on two signed integers. An example of logical formula is that used in (Dietz et al., 2015) to evaluate an n-bit addition operation on two's complement integers (s_1 and s_2) overflows is the following expression:

$$((s_2 > 0) \wedge (s_1 + s_2 > \text{INT\_MAX})) \\ \vee ((s_2 < 0) \wedge (s_1 + s_2 < \text{INT\_MIN}))$$

meaning that a signed addition can overflow if and only if this expression is true.
Regarding the example given in Figure 1, if we consider the conditional statement (line 7), an appropriate property to analyze this statement would be a formula that deals with with multiplication on two's comple-

[1] https://wiki.sei.cmu.edu/confluence/display/c

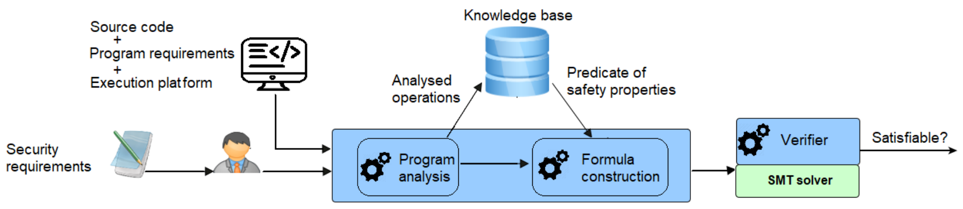


Figure 3: End-to-end approach.

ment n-bit signed integers, and also with the comparison with 0. The useful assertion that we propose is:

$$s_1 \times s_2 = 2^n \times k \wedge k > 0 \Rightarrow s_1 \times s_2 = 0$$

As pointed out in (Dietz et al., 2015) the result of an integer multiplication can wrap around many times. Consequently, a processor typically places the result of an n-bit multiplication into a location that is $2n$ bits wide. If we suppose that the result $s_1 \times s_2$ will occupy the high-order n bits, the integer is then evaluated by:

$$s_1 \times s_2 = \sum_{i=n}^{2n} x_i 2^i = x_n 2^n + x_{n+1} 2^{n+1} + \ldots + x_{2n} 2^{2n}$$

Let m be the first least significant non-zero bit in this expression, so the result can be written as:

$$s_1 \times s_2 = 2^m + \ldots + x_{2n} 2^{2n-m}$$

Naturally, if $m \geq n$, we have $s_1 \times s_2 = 0$. Moreover, the result can be rewritten as:

$$s_1 \times s_2 = 2^m \times \underbrace{(1 + \ldots + x_{2n} 2^{2n-m})}_{\text{k}}$$
$$= 2^n \times 2^{m-n} \times \underbrace{(1 + \ldots + x_{2n} 2^{2n-m})}_{\text{k}}$$

such that $k > 0$. More simply, the result of the multiplication becomes:

$$s_1 \times s_2 = 2^n \times k$$

with $k > 0$. Therefore, we get the proposed formula:

$$s_1 \times s_2 = 2^n \times k \wedge k > 0 \Rightarrow s_1 \times s_2 = 0$$

News feeds will be provided throughout the lifetime of the database. It will gradually populated by logical formulas extracted from software vulnerabilities directories and coding conventions. Basically, this task requires an expert in logics to build such formulas.

3.2 Model Construction

– The developer(s) provides the source code of the program to be analyzed. The program written in a specific programming language, in this work we focus on programs written in C language.

– He provides both the features about the execution environment and the platform on which the program will be compiled. In other words, all information about the targeted execution environment, e.g. operating system, architecture 32/64 bits and compiler. This information should guide the search for suitable formulas to pick from Knowledge Base.

– He can also provide other specific requirements, as requirements that specify constraints on the domain of variables and on the data structures appearing in the program. Referring to our example (given in Figure 1). In addition to the specification saying that "*the value of the userId of the administrator is equal to 0*", we also add two other constraints defining the variables domain: *userId* $\in [0, \ldots, 150 \times 10^6]$ and *serviceId* $\in [1, \ldots, 64]$.

– The developer(s) or the security expert(s) specifies a security requirement which is a statement of required security functionality that the program should satisfy. Often, security requirements are presented in an informal style, so their interpretation and implementation require some expertise. We aim at reducing the security expert burden to a minimum. Static analysis can be used to identify the program points (data/instruction) that can be affected by the security requirement. Consider the guideline stating "*Only the administrator user will access services with read/write privileges*". It represents a typical security requirement and means that the access to the secure areas is granted only to the administrator. By referring to our example it can be easily noticed that the statement at line 8 is concerned by this requirement.

We aim at constructing a model including all the parties involved in the analysis: the program and its context. The outcome of this modelling step is a logical formula that specifies both the program, the requirements, and the execution environment.

– For this an intermediate representation is built which in the Program Dependence Graph (PDG) augmented with information and details obtained from deep dependency analysis on the program (as shown in Figure 4). Static analysis tools (as Frama-C for analysis of program C analysis (Cuoq et al., 2012) and JOANA (Graf et al., 2013) for program

Java) can be used to construct such structure that capture control and (explicit/implicit) data dependencies between program instructions, and which constitutes a strong basis to perform a precise analysis. Figure 4 shows the PDG graph for the sample code given in Figure 1. Strong edges represent the control flows, the dashed edges refer to explicit and implicit data flows.

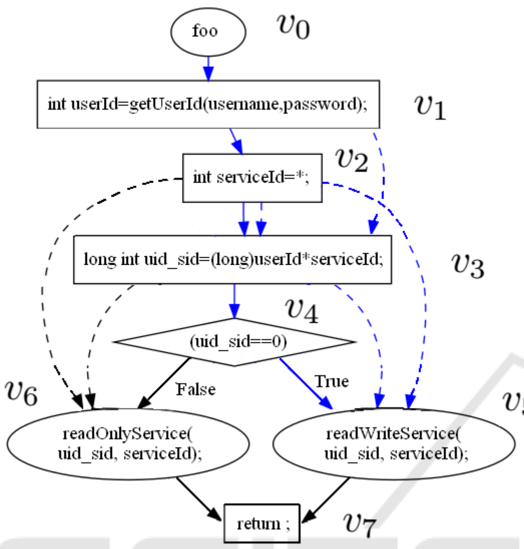

foo
v_0
int userId=getUserId(username,password);
v_1
int serviceId=*;
v_2
long int uid_sid=(long)userId*serviceId;
v_3
(uid_sid==0)
v_4
False
True
v_6
v_5
readOnlyService(uid_sid, serviceId);
readWriteService(uid_sid, serviceId);
return ;
v_7

Figure 4: PDG model for the sample code given in Figure 1.

– Based on security requirement and by relying on the assistance of security expert(s) we get the sensitive data. The exploring of PDG graph and tracking all dependencies (explicit/implicit) flows, we compute the set of all the sensitive information in a program. From a developer perspective, it is tough to fulfil this task without automatic tool, the complexity of this operation increases with the complexity and the program size. Referring to our example (Figure 1), sensitive data are clearly *password* and *username*. The complete exploration and computation will expand this set and produces a broader set: $\{uid\_sid, serviceId, userId, password, username\}$.

– Based on this complete set of sensitive data we identify on the PDG graph the set of critical statements (nodes). These nodes represent all the program points that may be potentially impacted by a security problem, regarding to the requirement under consideration. Referring to the intermediate representation of our example (Figure 4) we can clearly see that the vertex v_5 corresponds to a critical statement node. Indeed, it is the only program point where administrators can have read/write access. From this node, we generate through symbolic execution path conditions. A set of all paths leading to the execution of the statement at this node. More specifically, consider $\pi_1,\ldots\pi_n$ all possible paths leading to a node v_k the aim is to build a formula that is a disjunction of all path conditions: $\varphi(v_i) = \varphi(\pi_1) \vee \ldots \vee \varphi(\pi_n)$. A path in the PDG is a sequence from the entry node to a given node (i.e. $\pi = v_0 \ldots v_k$). If we denote by $C(v_i)$ the necessary condition for the execution of v_i meaning that the instruction at the node v_i can be executed only if $C(v_i)$ is satisfied. So the path condition can be rewritten as: $\varphi(v_i) = \bigvee_{1\leq i\leq n} \Big(\bigwedge_{0\leq j\leq k} C(v_j)\Big)$. Let us turn to our example, the condition for execution the statement at the vertex v_5 is $C(v_5) \triangleq uid\_sid = 0$. As the reader can notice, there is only one execution path leading to this node (naturally, one should also consider the control flow). So we get $\varphi(v_5) \triangleq (userId \times serviceId = 0)$. Afterwards, we formulate the program constraint that made up of path conditions formula plus the program requirements. We get:
$PC \triangleq (userId \times serviceId = 0) \wedge (serviceId \geq 1 \wedge serviceId \leq 64) \wedge (userId \geq 0 \wedge userId \leq 150 \times 10^6)$

– Considering the path condition, the next step involves seeking out execution context formula (EC) from Safety Knowledge Base. It contains a set of formulas that map software constructors with execution configurations, including operating systems, compiler settings, build process tools, etc. The question is, how to identify the relevant formula to pick? This is guided by the program constraint and the target architecture. For instance, for our example we clearly identified that the hotspot is the multiplication operation whose result is 0. By seeking out formulas that include the multiplication operation (and possibly concerning the targeted architecture), we will find out the formula already inserted:

$$EC \triangleq s_1 \times s_2 = 2^n \times k \wedge k > 0 \Rightarrow s_1 \times s_2 = 0$$

Actually, this formula should be instanced by the target architecture settings, i.e. substituted by 32 or 64 depending on target architecture.

– It remains now only to formulate the security constraint. Without a thorough examination and understanding of the given security requirement, this can be reformulated as follows: can a user who is not an administrator obtain a privileged service? We formulated the fact that a user is not an administrator with $userId \neq 0$. So get:

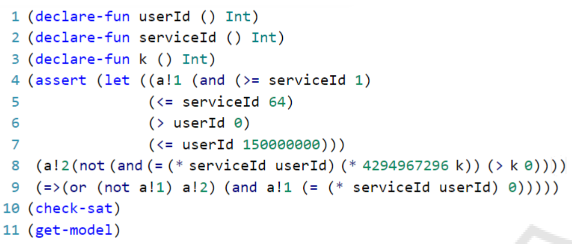
$$SC \triangleq userId = 0$$

The security problem to be solved is thus formulated as:
$(s_1 \times s_2 = 2^n \times k \wedge k > 0 \Rightarrow s_1 \times s_2 = 0) \vdash$

$(userId \times serviceId = 0) \wedge (serviceId \geq 1 \wedge serviceId \leq 64) \wedge (userId \geq 0 \wedge userId \leq 150 \times 10^6) \wedge (userId \neq 0)$

3.3 Checking Satisfiability

This step aims at verifying the satisfiability of the constructed model. We used an SMT (Satisfiability Modulo Theories) solver that is a powerful tool for checking satisfiability and supports arithmetic and decidable theories. We used Z3 prover developed by Microsoft Research (as shown in Figure 5).



```
(declare-fun userId () Int)
(declare-fun serviceId () Int)
(declare-fun k () Int)
(assert (let ((a!1 (and (>= serviceId 1)
              (<= serviceId 64)
              (> userId 0)
              (<= userId 150000000)))
 (a!2(not (and (= (* serviceId userId) (* 4294967296 k)) (> k 0))))
 (=>(or (not a!1) a!2) (and a!1 (= (* serviceId userId) 0)))))
(check-sat)
(get-model)
```

Figure 5: Specification Formula encoded in Z3.

By instantiating our formula by considering architecture of 32 bit we get the following model (valuation that satisfies the formula):

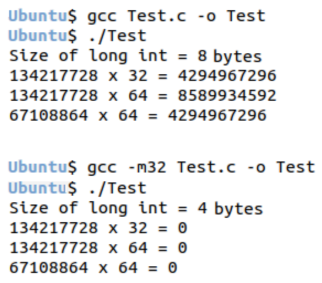
```
sat (model
  (define-fun k () Int 2)
  (define-fun userId () Int  134217728)
  (define-fun serviceId () Int 64))
```

meaning that the execution of our program on this architecture can violate the security requirement. If the identifier of the user (*userId*) equals 134217728 requests a service (*serviceId*) equals 64 we may have a security problem (take on the role administrator). In order to get **all possible models**, we used python script in which the formula was updated by negating each model found till the formula became unsatisfied. So we found all problem cases:

```
sat [serviceId=32, userId=134217728, k=1]
sat [serviceId=64, userId=67108864, k=1]
sat [serviceId=64, userId=134217728, k=2]
```

For practical validation of these results, we executed our example on an *Ubuntu 18.04 , 64 bit*: we run binary optimized for both 32 and 64 bit architecture. The binary optimized for 32 bit architecture can be obtained by adding a flag "-m32" while compiling C source otherwise it is optimized for 64 bit. Figure 6 shows the compilation mode and the output of the execution of the resulting program.

So there are no surprises there, experimentation reinforces the theoretical results. Consequently, we focused our efforts on studying the effects of various architectures (operating systems and compiler options) on integer error classes, more specifically, on arithmetic overflow caused by the multiplication operation. The results of the evaluation is summarized in Table 1. We report the occurrence of security violation (✓) and its absence (✗) for each configuration.



```
Ubuntu$ gcc Test.c -o Test
Ubuntu$ ./Test
Size of long int = 8 bytes
134217728 x 32 = 4294967296
134217728 x 64 = 8589934592
67108864 x 64 = 4294967296

Ubuntu$ gcc -m32 Test.c -o Test
Ubuntu$ ./Test
Size of long int = 4 bytes
134217728 x 32 = 0
134217728 x 64 = 0
67108864 x 64 = 0
```

Figure 6: Practical validation of theoretical results.

Discussion. Naturally in early stages of this work we attempted to carry out the analysis by using **Frama-C** (Cuoq et al., 2012), which is a publicly available and well-known toolset for analysis of C programs; which furthermore supports variation domains for variables by means of **Eva** plugin. To encode function `getUserId`, we used the function random that returns a value within the domain $[0,\ldots,150 \times 10^6]$. We encoded function `read` in a similar way. To check whether the targeted security property is satisfied or not, we inserted the following assertion in the header (requires clauses):

```
//@ assert userId==0;
```

indicating that only an authorized user (an administrator) can execute the first branch. We faced with an issue, Frama-C uses a constant $fc\_rand\_max = 32767$ that bounds the returned values of random function call. For circumventing this problem, we encoded the range of values of the domain in an array. Although, the tool inserts a clause in the program, the latter is not relevant, it is not linked to a particular architecture. Indeed, software security analysis in most existing tools are performed, regardless to a particular execution architecture. Nowadays, it is known that vulnerabilities can be inadvertently introduced by the execution environment for various reasons, typically they can be induced by compiler settings [2].

4 RELATED WORKS

Assessing security properties of software components relying on formal approaches is an active area of

[2]https://software.intel.com/content/www/us/en/develop/articles/size-of-long-integer-type-on-different-architecture-and-os.html

Table 1: Effects of Various Architectures on Arithmetic Overflow.

Target architecture x86_64			
Target OS	Compiler	without flag -m32	with flag -m32
Windows10 64	gcc10.2.0	✓	
	clang11.0.0	✓	✓
Ubuntu18.04 64	gcc7.5.0 clang6.0.0	✗	✓
Ubuntu 14.04 32	gcc4.8.2	✓	✓
OS X 64	clang11.0.0	✗	✓

research. In contrast, to model-driven engineering (or security by design) approach offering methodologies for designing secure system e.g. (Ameur-Boulifa et al., 2018), in this paper we focus on the security by certification approach, we described how security vulnerabilities and architectural features might be captured by security experts and verified formally by developers. We focus on errors induced by integer overflow.

Several works, e.g. (Wagner et al., 2000; Dietz et al., 2015) aim to a better understanding integer overflow in programming language, and extract assertions from a source code. In (Dietz et al., 2015), authors have demonstrated that the analysis of integer can be driven by using logical expressions as precondition tests to include in the C/C++ source code and following the code execution check whether post-condition CPU flags are set appropriately. An active research is done to improve software security aspect by identifying potential vulnerabilities vulnerabilities. Some of them are based on static analysis (Han et al., 2019; Aggarwal and Jalote, 2006), dynamic analysis (Aggarwal and Jalote, 2006) and symbolic execution (Zhang et al., 2010; Li et al., 2013; Boudjema et al., 2019).

In (Zhang et al., 2010), authors present a security testing approach based on symbolic execution. The efficiency of this approach depends on the relevancy of the set of test cases given as entries. An improved approach is proposed in (Li et al., 2013), without taking into account test cases. A forward and backward analysis are performed to detect vulnerable instructions and construct data flow trees based on sensitive data. The path exploration problem is controlled by considering only execution paths related to vulnerabilities. The approach presented in this paper is the closest to our approach. However the approach presented in this paper does not consider vulnerabilities that may be induced by the execution environment.

Some work focused on binary code. Specifically in (Boudjema et al., 2019), authors detect exploits by combining concrete and symbolic execution with sensitive memory zones analysis. A fixed-length execution traces annotated by sensitive memory zones are considered, based on a limit execution time of the given code, so vulnerabilities that arise after a long execution time are not detected. The approach proposed in (Wang et al., 2008) to detect vulnerabilities caused by buffer overflow on binary code relies on symbolic execution and satisfiability analysis. It uses also a technique for automatically bypassing some security protections. In (Han et al., 2019), vulnerabilities are checked considering only those matching given behaviour patterns. A control flow graph is computed, then the corresponding vulnerability executable path set is defined considering only paths with vulnerability nodes. This method strongly depends on the relevancy of the patterns used, while in our approach all predicates saved in the knowledge database can be used to catch vulnerabilities.

From the perspective of tools, several have been developed to detect vulnerabilities on source or binary code e.g. in (Wang et al., 2008; Boudjema et al., 2019; Han et al., 2019; Ognawala et al., 2016). For example, the tool developed in (Ognawala et al., 2016) combines symbolic executions and path exploration to detect exploits on binary code. It is also able to assign severity levels to reported vulnerabilities.

5 CONCLUSION AND FUTURE WORK

The security aspect of critical programs can be improved by eliminating vulnerability bugs, it helps to increase the system robustness against attacks. In this paper, we proposed a formal-based approach to detect security vulnerability induced by integer errors. One major advantage of our approach, it relies on models describing both, a system's architecture and software in integrated formulas. Furthermore, security analysis is conducted via combination of symbolic execution and satisfiability analysis to check vulnerability bugs caused by an unsafe programs, and even those introduced inadvertently by the execution environment. We analyzed program source code to identify the critical instructions or operations, which can be exploited to cause an unsafe memory behaviour. For that, a safety knowledge base is constructed using a formal representation of errors and recommendation reports and also requirement that can provided by users. The created safety knowledge base is used to improve the obtained formulas by adding safety constraints. The latter are checked by an SMT solver to detect vulnerability bugs. We illustrate our approach through an example. Definitely the efficiency of our approach depends strongly on the relevancy of the used safety knowledge base. The current work addresses the problem of extracting and using knowledge of attack and software vulnerabilities directories to build safety properties patterns. An alternative solution would be to use interactive annotation through an interface for the tool as it has been done in (Thomas, 2015). The aim of our ongoing and future work is to formalize more unsafe predicate including pointer references, cast operations and more. By expanding the knowledge data base, we can treat a broader class of bugs, and can then identify more vulnerabilities.

REFERENCES

2019 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

CERN Computer Security. https://security.web.cern.ch.

Aggarwal, A. and Jalote, P. (2006). Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference*, volume 01, pages 343–350, USA. IEEE Computer Society.

Ameur-Boulifa, R., Lugou, F., and Apvrille, L. (2018). SysML model transformation for safety and security analysis. In *ISSA 2018:International workshop on Interplay of Security, Safety and System/Software*, Spain. ACM IPCS.

Boudjema, E. H., Verlan, S., Mokdad, L., and Faure, C. (2019). VYPER: Vulnerability detection in binary code. volume 3. Wiley.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-C - A software analysis perspective. In *10th International Conference, SEFM 2012, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer.

Dietz, W., Li, P., Regehr, J., and Adve, V. (2015). Understanding Integer Overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.*, 25(1).

Graf, J., Hecker, M., and Mohr, M. (2013). Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Software Engineering 2013, Aachen*, volume P-215 of *LNI*, pages 123–138. .

Han, L., Zhou, M., Qian, Y., Fu, C., and Zou, D. (2019). An optimized static propositional function model to detect software vulnerability. *IEEE Access*, 7:143499–143510.

Hohnka, M. J., Miller, J. A., Dacumos, K. M., Fritton, T. J., Erdley, J. D., and Long, L. N. (2019). Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems*, 16(10):409–426.

Kosmatov, N. and Signoles, J. (2013). A lesson on runtime assertion checking with Frama-C. In *4th International Conference Runtime Verification RV 2013, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 386–399. Springer.

Li, H., Kim, T., Bat-Erdene, M., and Lee, H. (2013). Software vulnerability detection using backward trace analysis and symbolic execution. In *International Conference on Availability, Reliability and Security, ARES 2013, Germany, September 2-6, 2013*, pages 446–454. IEEE Computer Society.

Ognawala, S., Ochoa, M., Pretschner, A., and Limmer, T. (2016). Macke: Compositional analysis of low-level vulnerabilities with symbolic execution. In *31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 780–785, USA. Association for Computing Machinery.

Seacord, R., Dormann, W., McCurley, J., Miller, P., Stoddard, R., Svoboda, D., and Welch, J. (2012). Source Code Analysis Laboratory (SCALe). Technical Report CMU/SEI-2012-TN-013, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Thomas, T. (2015). Exploring the usability and effectiveness of interactive annotation and code review for the detection of security vulnerabilities. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2015, USA, October 18-22, 2015*, pages 295–296. IEEE Computer Society.

Van der Veen, V., Dutt-Sharma, N., Cavallaro, L., and Bos, H. (2012). Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses*, pages 86–106, Berlin, Heidelberg. Springer Berlin Heidelberg.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17.

Wang, L., Zhang, Q., and Zhao, P. (2008). Automated detection of code vulnerabilities based on program analysis and model checking. In *8th International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, China*, pages 165–173. IEEE Computer Society.

Younan, Y., Joosen, W., and Piessens, F. (2004). Code injection in C and C++:A survey of vulnerabilities and countermeasures. Technical report, Department Computer Wetenschappen, Katholieke Universiteit Nl Leuven.

Zhang, D., Liu, D., Lei, Y., Kung, D. C., Csallner, C., and Wang, W. (2010). Detecting vulnerabilities in C programs using trace-based testing. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, USA, June 28 - July 1 2010*, pages 241–250. IEEE Computer Society.