VYPER: Vulnerability detection in binary code

El Habib Boudjema, Sergey Verlan, Lynda Mokdad, Christèle Faure

VYPER: Vulnerability detection in binary code

El Habib Boudjema[1,2], Sergey Verlan[1] Lynda Mokdad[1], and Christèle Faure[2]

[1] LACL, Université Paris-Est, Créteil, France
[2] Saferiver, Montrouge, France

Abstract. This paper presents a method for exploitable vulnerabilities detection in binary code with almost no false positives. It is based on the concolic (a mix of concrete and symbolic) execution of software binary code and the annotation of sensitive memory zones of the corresponding program traces (represented in a formal manner). Three big families of vulnerabilities are considered (taint related, stack overflow and heap overflow). Based on the *angr* framework as a supporting software VYPER was written to demonstrate the viability of the method. Several test cases using custom code, Juliet test base and widely used public libraries were performed showing a high detection potential for exploitable vulnerabilities with a very low rate of false positives.

Keywords
Static analysis, Binary code analysis, Exploitable vulnerability detection, Symbolic execution, Exploitable flaws, x86_64 assembly security.

1 Introduction

Building robust and secure software free from vulnerabilities is becoming a major concern of the IT community. Programming errors committed at the development stage are the main source of vulnerabilities and security holes. Different strategies [13], [9], [10], methods [5], [8], tools [14], [16], [18] have been and are being developed to detect these programming errors or to prevent them from happening. Software code analysis is one of these methods aimed to detect vulnerabilities early in software life cycle. This method has many drawbacks. We cite the problem of "false positives" where the analysis tool detects vulnerabilities, which in fact are not existing. Software security analysts will have a harsh task to sort true and false positives especially for large and complex software, and they may miss severe true vulnerabilities.

In this paper we propose a method that describes how to detect exploitable vulnerabilities with almost no false positive. The given solution also permits to a software analyst to easily confirm the vulnerability by providing him an input sample that can trigger it. Our solution can also be used to automatically sort true and false positive vulnerabilities obtained from other software analysis tools.

The proposed method is a 3-stages process that first computes program traces by a concolic (a mix of concrete and symbolic) execution of the software binary code. Next, for each state of the computed trace(s) an annotation of sensitive memory zones is computed. Finally, for each state, a predicate based on this

annotation and the result of the symbolic execution allows to verify if a vulnerability can be triggered in the corresponding state (giving the necessary input as well). The use of binary code instead of source code is motivated by the fact that it contains all the necessary details to accurately find exploitable vulnerabilities. Such approach best responds to the objective of generating no false positive. We concentrated on three classes of vulnerabilities: taint related, stack overflow and heap overflow and we gave corresponding descriptions of the annotation and detection functions.

Concolic execution is a software analysis technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs). It was introduced by Godefroid et al.[6] where it was used to assist random testing to cover a maximum numbers of execution paths. This method is also used by KLEE [2] for their unassisted high coverage testing. In their tool AEG (Automatic Exploit Generation) Avgerinos et al.[1] were interested in automatically generating an exploit by combining source analysis to find the exploitable vulnerability and binary analysis to produce the exploit. In our work we focus more on finding different classes of vulnerabilities, we do not generate a completely working exploit but only the inputs necessary to trigger the first stage of the vulnerability. The same difference exits between our approach and Mayhem [3] that is able to automatically find a vulnerability and generate an exploit based only on binary analysis. Different tools and methods were developed since the introduction of concolic execution. Shoshitaishvili et al.[7] describes almost all state of art techniques used in binary analysis based on concolic execution and other techniques that they implemented in the open source *angr* framework [12].

To demonstrate the viability of our approach we developed a real testable tool Vyper (VulnerabilitY detection based on dynamic behavioral PattErn Recognition) that uses *angr* framework. The tool can be used in "search mode" to search for vulnerabilities. In what we call "refine mode" the tool is used to check if a given vulnerability is real or not and to produce the input that permits to confirm it. We tested Vyper on several custom test cases, Juliet test suites [17] and some widely used open source libraries (openssl, libpng and libtiff). The results are very promising: in the first two cases most exploitable vulnerabilities were detected with almost no false positive (2% versus 20% when using other tools), while in the last case several previously unknown vulnerabilities were detected.

The paper is organized as follows: we define what is meant by *exploitable vulnerabilities* in Section 2. We give the formalization of the proposed solution in Section 3.1. Where in Section 3.2 we show how this formal model can be applied on examples of exploitable vulnerabilities. The Sections 4 and 5 detail the implementation and evaluation of Vyper. We conclude and discuss the future works in Section 6.

2 Exploitable vulnerabilities

In this paper we consider exploitable vulnerabilities, *i.e.* vulnerabilities that allow an attacker to execute an arbitrary code or read data at an arbitrary memory location. Executing an arbitrary code means that an external application user is able only via input vectors (command line arguments, environment variables, file system, network sockets) to hijack the application control flow and execute code fed as input to the application. Reading data at an arbitrary location allows an external user to obtain via the application output confidential information (passwords, encryption keys etc) or gain knowledge about the memory layout (stack and heap addresses, dynamic library loading addresses). This knowledge can help an attacker to bypass some efficient application protection mechanisms such as the ASLR (Address Space Layout Randomization) or DEP (Data Execution Protection), allowing him to exploit software running on recent and hardened operating systems. Others vulnerabilities such as application crashing or interference with application logic etc. will not be covered in the paper.

We focused our study on the most commonly exploited vulnerabilities in wild. We consider the following classes of vulnerabilities:

1. *Taint related vulnerabilities*:
   Format string, or command injection vulnerabilities, that are caused by calling some dangerous functions (printf, syslog, system, execlp) with a tainted attacker supplied argument. These vulnerabilities are exploitable if the tainted argument has not been sanitized at all or has been incorrectly sanitized.
2. *Stack overflow*:
   A stack overflow occurs when data is written or read at a location that is beyond the stack buffer maximum size. A stack overflow will be exploitable if the return value stored on the stack is erased with an attacker supplied value allowing him to hijack the control flow into a desired location. Note that not all stack overflows are exploitable, for example if we erase only few stack memory locations and we do not reach the stored return address [19], the caused stack overflow is not exploitable. The Fig. 1 depicts a case of an exploitable stack overflow on an X86_64 machine running on Unix based OS.

3. *Heap overflow*:
   When a heap buffer is allocated the dynamic memory allocator adds different control data before and after the buffer. Exploitable heap buffer overflow occurs when a heap is written beyond its size with a user supplied value (value derived form the input). Because the erased control data will confuse the memory allocator, this can potentially allow the attacker to execute arbitrary code. A *use-after-free vulnerability* occurs when a freed buffer is accessed and can be exploited in the same manner. *Double freeing* a heap buffer can also allow an attacker to obtain a control hijack and execute an arbitrary code. Exploiting a heap buffer overflow is less easy than stack overflow. It needs a deep knowledge of heap meta-data and heap allocator

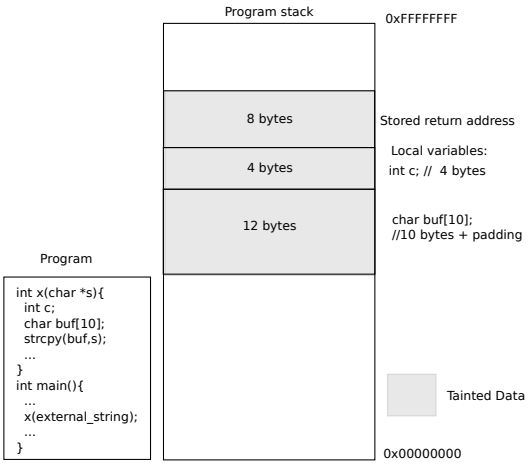

Fig. 1. Exploitable stack overflow : external_string containing 24 bytes of tainted data is written to the buffer erasing the return address

algorithms. In practice heap overflow exploits are probabilistic and use some techniques such as heap spraying [15] and heap layout information leak to make the exploits more reliable. The Fig. 2 depicts a case of an exploitable heap overflow where a tainted is written to *buf1* and the write operation overflows and erases the metadata of the following buffer *buf2* leading to an exploitable heap overflow situation.

3 Exploitable vulnerabilities detection

From the examples above we observe a general pattern shown in the Fig. 3: a vulnerability is exploitable into a control hijack, if some *sensitive* memory zones (function argument, stored return address, heap buffer start and end) are accessed (read, write) in some special situations (with data derived from input or with data not correctly sanitized). By searching for this pattern in analyzed applications we will be able to locate *exploitable* vulnerabilities. Finding these exploitable vulnerabilities is done in 3 steps:

- *Program traces construction:* computed using concolic execution, that guarantees the propagation of initial symbolic inputs along the generated traces with path formulas expressed within these inputs. We note that we construct all traces of a maximum fixed length to guarantee the termination of this computation.
- *Sensitive memory annotation:* in this step we will construct sensitive memory zones that will be used for checks.
- *Vulnerability detection:* in this step we will check if the operation we are executing in the corresponding program state cause a sensitive memory zone

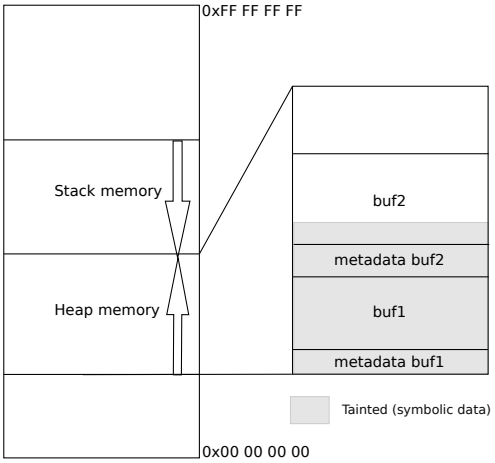

Fig. 2. Exploitable heap overflow : Overflowing buf1 will erase "metadata buf2" and allow an arbitrary code execution

to be written with data coming form input vectors (symbolic data). As result the corresponding vulnerability information is reported as well as the input data that will trigger this vulnerability (if possible).

In what follows we will present the formalization of the concept of the program and sensitive memory and next we will describe the detection of vulnerabilities presented in Section 2 in terms of that model.

3.1 Formalization

Below, we present a formalization of the method detecting and reporting exploitable vulnerabilities based on their general behavioral pattern.

To detect an exploitable vulnerability in a given program we propose a method based on the annotation of program traces states' space. As in model checking methods, we will construct reachable program states and keep the execution traces. After that, for each trace we will annotate each state by a list of sensitive memory zones, based on the executed instructions. Finally, for each state we will report vulnerabilities based on checks and constraint resolution done on the program state, execution trace and the constructed annotation containing sensitive memory zones.

We notice that for our goal we do not need to distinguish between memory locations and registers, so we can assume that the starting addresses of the memory correspond to the registers.

We will abstract the real computer program and the concrete machine model by random-access machine (RAM) model [4] enriched with additional instructions.

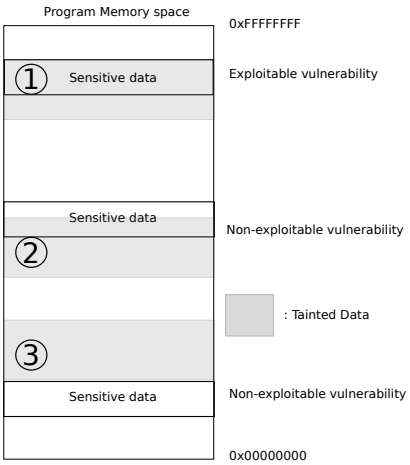

Fig. 3. Exploitable Vulnerability pattern : (1) Sensitive memory is erased with tainted data; (2) sensitive memory is partially erased; (3) sensitive memory is not erased. The vulnerability is exploitable only in case 1.

In order to be closer to the reality we consider two restrictions on the RAM model: first we assume that the number of registers (the memory) is finite and of size M; second we consider that each register (memory cell) can hold a bounded integer value – at most 2^n. Hence, the memory can be defined as a vector of size M: $Memory \in INT(n)^M$, where $INT(n) = \{0, \ldots, 2^n - 1\}$ be the set of numbers that can be represented with at most n bits in the binary notation.

Next, we augment the RAM model with additional instructions. An instruction is defined by the operation and the list of its arguments that are indexes of memory locations on which the operation is executed. We remark, that we consider instructions using constant values as new types of instructions. So, let OP be the set of all possible operation codes, then an instruction is a tuple $I = (op, a_1, \ldots, a_k)$, where $op \in OP$ and k and a_k depend on op: k is the number of operands of op and a_k are the corresponding operands. The set of all instructions will be denoted as $INS \subseteq OP \times \mathbb{N}^K$, where K is the maximal number of arguments for any instruction.

A program $P = I_1, \ldots, I_n$ is an ordered list of instructions: $P \in INS^*$.

A state of the machine is given by the contents of the memory and by the current instruction index: $S_j = (i_j, M_j)$, $1 \leq i_j \leq n$, $M_j \subseteq INT(n)^M$. Suppose that an execution of an instruction I_{i_j} allows to pass from state S_j to S_{j+1}: $S_j \overset{I_{i_j}}{\Longrightarrow} S_{j+1}$, where the memory contents is updated accordingly.

A trace (an execution) of length n is the following sequence starting in the initial state S_0

$$\pi : S_0 \overset{I_{i_0}}{\Longrightarrow} S_1 \overset{I_{i_1}}{\Longrightarrow} \ldots \overset{I_{i_{n-1}}}{\Longrightarrow} S_n.$$

We annotate each state S_i of trace π. We denote by A_i^π the corresponding annotation. An annotation is a list of triples $(memLocation, size, attribute)$, where $memLocation, size$ are respectively the address and the size of the annotated memory. The *attribute* value is used to keep the type of the corresponding memory zone ($RETURN\_ADDRESS, HEAP\_METADATA$, $CALL\_ARGUMENT, etc.$). Hence $A_i^\pi \in (\mathbb{N} \times \mathbb{N} \times Attribute)^*$. We will omit the superscript π if π can be deduced from the context.

For each state S_i of π we consider the detected vulnerabilities information denoted V_i^π, which is a list of couples $(codeLocation, VULN\_INFO)$, where $codeLocation$ is the instruction offset in the code and $VULN\_INFO$ is a structure that contains the name of the vulnerability, the corresponding CWE [11] identifier and the context (call stack, input values, etc). Hence, $V_i^\pi \in (\mathbb{N} \times VInfo)^*$. As above, we will omit the superscript π if it can be deduced from the context.

We use a *concolic* execution of the program. One of the major advantages of such method is that it keeps track of the input data called symbolic data, so it is possible to distinguish if a value of a memory location is computed using external data. As we could see above, the input data plays a major role to make a vulnerability exploitable.

Another important point is that we limit the execution traces to a certain length. The idea behind this limitation is that we would like to detect vulnerabilities in a reasonable time, implying that the execution of the program will be stopped after some number of steps. The drawback of this approach is that the vulnerabilities requiring a higher number of steps for the detection will not be found. From the other point of view, the tool we use for the concolic execution already has similar limitations.

Hence, we will consider execution traces of length at most L and we will denote the corresponding set by Π_L. We would like to remark that below we will consider that Π_L is already computed (by making all corresponding runs). However, in the implementation we have chosen another approach where at each step we keep track of all traces and we evolve them in parallel. While this leads to the same final result, conceptually, it is easier to suppose that the set of traces is already computed.

For every program trace $\pi : S_0 \Longrightarrow \dots S_n$ of size n we will populate the list A_i, $0 \leq i \leq n$ as follows:

$$\begin{aligned} A_0 &= \emptyset \\ A_{k+1} &= Annotate(S_k, A_k, I_{i_k}), \quad 1 \leq k \leq n. \end{aligned}$$

The function *Annotate* will permit to store the access to sensitive memory zones and this information will be further used for the detection of different kind of vulnerabilities. Based on the general behavioral pattern of a vulnerability some special memory locations have an important role, for example the return addresses stored in the stack frame or the buffer meta-data stored on the heap. When these memory locations are written, corresponding states need to be annotated for a future search for a vulnerability.

Next, we compute all V_k, $0 \leq k \leq n$ as follows:

$$V_0 = \emptyset$$
$$V_{k+1} = Detect(S_k, A_k, I_{i_k}, V_k), \quad 1 \leq k \leq n.$$

The function *Dectect* checks for different vulnerabilities. The check is based on the annotated list for each state. Given the behavioral pattern of a vulnerability and the previous annotations on the execution path the *Dectect* function will check if the conditions to have an exploitable vulnerability are met. In the positive case, it stores the information about the found vulnerability in V_{k+1}. For example if a memory location is annotated as being a stored return address in a function stack frame and it is written with tainted (symbolic) data, then we signal this as exploitable stack overflow.

The introduced model allows to describe a general detection framework based on the annotation of used memory locations. In the next section we describe how this framework can be applied for the classes of vulnerabilities we are interested by.

3.2 Formal model application on exploitable vulnerabilities

For practical reasons we will group program instructions into functional groups (i.e. memory access, subroutine call), as for many instructions the *Annotate* and *Detect* functions are almost identical. These groups are closely related to the functioning of *angr* framework [12] and especially to event based breakpoints that can be fired, e.g. on a memory/register access or a function call. The used part of [12] will be detailed in Section 4.3. However, it should be clear that it makes no particular difficulty to unroll the corresponding definitions and give them for concrete instructions.

So, below we present the functions *Detect* and *Annotate* used for the detection of the 5 types of vulnerabilities discussed in this paper.

- Taint vulnerabilities (printf, system):
  - Function *Annotate*:
    It is the identity function so the annotation list A_i will be always empty.
  - Function *Detect*:
    If the current operation is a call to taint related function, and the format argument is pointing an symbolic area then report a vulnerability.
- Stack overflow:
  - Function *Annotate*:
    If the current operation is a function call, then add the triplet $(stack\_pointer, reg\_size, STACK)$ to the list A_i. If it is a `ret` operation subtract the corresponding stack pointer element from A_i.
  - Function *Detect*:
    If the current operation is a memory write and the destination argument is within A_i and marked as $STACK$ and the source is symbolic then report an exploitable stack overflow vulnerability.

- Heap overflow:
  - Function *Annotate*:
    if the current operation is a call to a memory allocation function then add the triplet $(malloc\_ret\_value, mallo\_arg, HEAP)$ to A_i
  - Function *Detect*:
    If the current operation is a memory write, and the data is written beyond the size of a buffer annotated as a $HEAP$ element and the source buffer is symbolic then report an exploitable heap overflow vulnerability.
- Double free:
  - Function *Annotate*:
    When *free* is called, the passed argument is annotated as being already freed buffer.
  - Function *Detect*:
    When *free* is called, we check if the passed argument is in our sensitive memory zones and marked as freed. In the positive case we report an exploitable vulnerability.
- Use after free:
  - Function *Annotate*:
    When *free* is called, the passed argument is annotated as being already freed buffer.
  - Function *Detect*:
    When a heap buffer is accessed we check if it is in the set of sensitive memory zones marked as already freed. If so, we report an exploitable vulnerability.

4 Implementation of VYPER

As stated before we built a tool called VYPER that implements the above behavior. Its implementation needs a good concolic execution engine. The implementation of a concolic engine from scratch is a hard task, which is not in our research scope. For this reason VYPER uses the *angr* framework [12] for program loading and states exploration. This allows to accelerate the development and permits us to focus more on vulnerability detection task. Another important point is that *angr* is actively developed and continuously improved, so our tool will benefit from further improvements of the framework.

4.1 A brief description of *angr* framework

angr is an open-source framework available to the security community. It is a binary analysis framework that implements a number of analysis techniques that have been proposed in the past. This allows researchers to use them without wasting their effort reinventing the wheel(s). We just cite some techniques implemented and documented in *angr* framework: binary loader for different OSes and architectures, control flow graph (CFG) computation, data flow graph (DFG) computation, value-set analysis (VSA), symbolic execution using execution path explorers, event-based breakpoints.

4.2 VYPER specification

VYPER can be specified as follows:

- Input: the binary program, the entry point function, analysis mode (optional, default:detection), vulnerability class (optional, default:all).
- Output: Vulnerability report containing for each reported vulnerability: vulnerability CWE identifier, location information, input values, call stack.
- Requirements:
  - Load the binary file and initialize the initial state as specified with a special input file if any.
  - Activate the requested checkers by setting the corresponding breakpoints that are used for the annotation and detection purposes.
  - Launch and control the concolic execution.
  - If an annotation breakpoint is hit, trigger callback function that will store the annotation information.
  - If a detection breakpoint is hit, trigger callback function that will check if the executed code is vulnerable, and store vulnerability information if a vulnerability is present.
  - If the symbolic execution is stopped (after a timeout) or finished, collect all the reported vulnerabilities and output it in the requested format.

4.3 VYPER implementation details

The tool contains about 600 lines Python code. Each vulnerability checker is implemented in a separate function and can be activated via command line arguments. We also implemented functions to pretty print the control graph flow of the analyzed application for debugging purposes. Due to some missing C library functions stubs in the framework, about 10 functions were implemented. The Fig. 4 shows the general algorithm followed by VYPER. In the following paragraphs we will explains the internals of each box.

1. *Load the program.* The binary of the analyzed program which is given as an argument is loaded by calling the *angr.Project* class. The option *auto_load_libs* is set to false. This will disable loading system libraries. All called system functions must be stubbed to have a correct analysis.
2. *Generate the CFG (Control Flow Graph).* CFG is generated based on the previously loaded program. We make use of *CFGFast* class of *angr*.
3. *Prepare program's initial state.* Before launching the analysis we initialize the analyzed program state by giving the following information:
   - Analyzed program parameters (argvs) and environment variables: they can be concrete values, symbolic values or a mix of both. All this information is specified in a file given as an argument to VYPER, or set to default values if no file is given.
   - Standard input, files, network sockets: to be initialized as specified by the user, or set to default symbolic values. The *angr* framework gives the possibility to use a concrete file system and directly interact with real files.

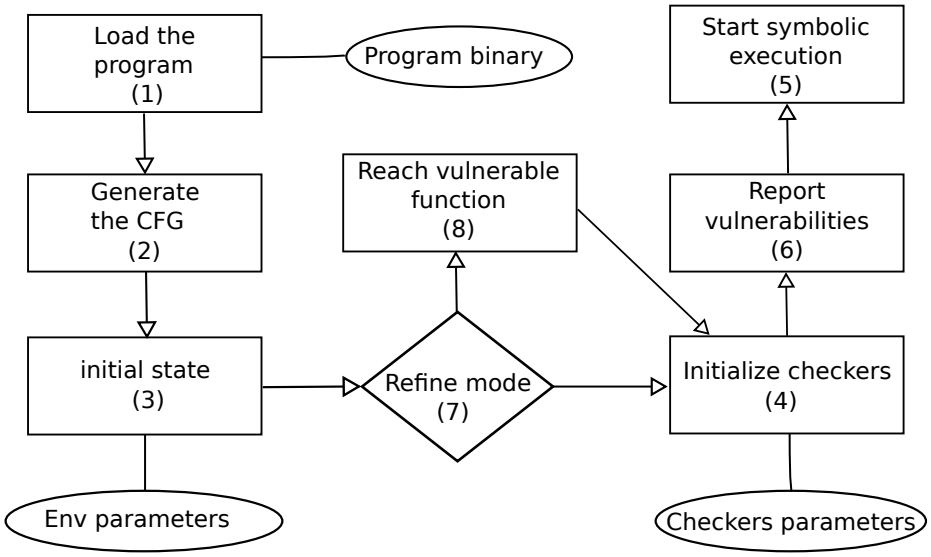

Fig. 4. VYPER general algorithm

- The program entry point: if we analyze a whole program and not only a part of it, this argument must not be set, letting *angr* guessing automatically the entry point directly from the binary headers.

4. *Initialize checkers.*
   Initializing checkers is done by inserting *breakpoints*, that, when hit, will trigger the convenient check routine, thus detecting the vulnerability and collecting necessary reporting information. For example, to detect a *tainted argument* we will insert a *breakpoint* that fires when a call to a vulnerable function (printf, fprintf etc) is performed. This breakpoint will call *check_tainted_arg* that will report a format string vulnerability. Checkers are all deactivated by default and activated only via the corresponding VYPER argument. This structure allows VYPER to be easily extended for new vulnerability checks by adding a breakpoint that will launch checks on program state and report the vulnerability if these checks pass. Since the checks can be activated or deactivated, the analysis can be faster if we are interested only in special category of vulnerabilities checked by VYPER. The detection procedure corresponding to vulnerabilities considered in this paper is depicted on Fig. 5
5. *Start symbolic execution.* In this step the symbolic execution is launched, and continued until all the CFG is covered. This is done via the *PathGroup* class of *angr* framework.
6. *Report vulnerabilities.* When vulnerabilities are detected they are not directly reported (only a small notification is emitted in the execution log). The found vulnerabilities and their related data (location, call stack, input values etc)

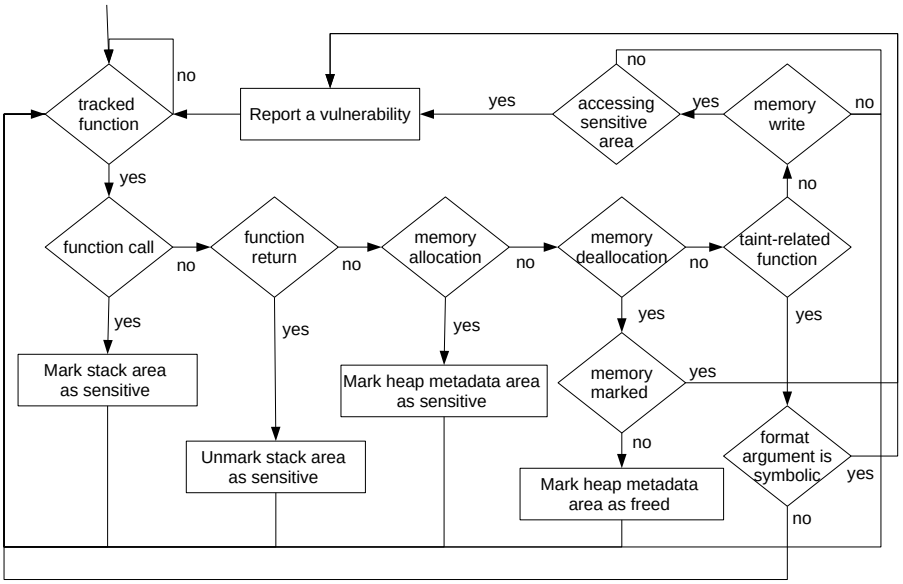

Fig. 5. The detection procedure for classes of vulnerabilities considered in this paper.

are stored in memory and reported in the requested format (txt,xml,html etc) at the end of the analysis.

7. *Refine mode.*
   VYPER can be used in *check mode* to find exploitable vulnerabilities or in *refine mode* to refine vulnerability reports obtained by some other tools in order to eliminate false positives. The corresponding reports are fed as input along with the program to analyze. This feature can allow an application auditor to specify vulnerabilities that should not occur at some code location, e.g. searching for exploitable stack overflows in authentication related code can be very useful to grant the security of the whole application.
8. *Reach vulnerable function*
   This part is executed when VYPER is running in *refine mode*. Using the *PathGroup* class of *angr* framework, the analyzed program is explored without any checker activated until the execution reaches the vulnerable function.

5 Testing and validation

To evaluate our implementation of the proposed method we used 3 types of test cases:

– **Custom test cases**: During the development, we built a test base that was used to test the detection of some simple cases of vulnerable code. This base was also used for debugging and non-regression purposes.

Table 1. Results on custom tests.

Vulnerability	CWE-ID	Comment
tainted format	134	The code contains a vulnerable call to printf conditioned with input values.
stack overflow	121	The code contains 2 stack overflows: one with a loop and an index the other with a call to strcpy.
double free	415	The test calls 2 times "free" in 2 different functions on the same buffer pointer
use-after-free	416	The test code tries to access a memory location of a freed buffer. The test is reported as a heap overflow.
heap overflow	122	A C++ program that allocates an array of objects and causes a write overflow.

- **Synthetic test cases**: This testing was done using publicly available Juliet test base [17] that contains thousands of test cases specially written to test static analysis software.
- **Real applications**: We used some well known applications and libraries that are available in open-source to test our tool on real life code.

5.1 Testing on custom test cases

Table 1 summarizes the different test objectives. All tests were run in detection mode and all of the introduced vulnerabilities were detected and reported correctly. For example, one of these custom tests makes several calls to *printf* function. All calls except one use valid constant format specifiers. The remaining call use a tainted format string and is only called if the input from *argv* has a special value. The vulnerable call was correctly detected and the needed input value was precisely reported.

5.2 Testing on Juliet test base

Juliet [17] is a collection of test cases written in C/C++ languages. It contains examples for 118 different CWEs [11]. A test case contains at least a vulnerable code (flaw) and the same code with the vulnerability fixed (fix). Flawed or fixed code can be activated using compiler macros. For each CWE, test cases are created using the simplest form of the flaw as well as other cases testing the same flaw with added control or data flow complexity. For example *CWE134_Uncontrolled_-Format_String__char_connect_socket_snprintf_01.c* test case will test the CWE 134 with tainted data source from "connect_socket" and a sink vulnerable function "snprintf". The suffix "01.c" means that this test is the simplest case of this flaw. A more complex version of this test case has the

name *CWE134_Uncontrolled_Format_String__char_connect_socket_snprintf_11.c* and is testing the same vulnerability but with more complex control or data flow, i.e. using intermediate variables or function calls crossing multiple files etc.

For our testing we have developed scripts that will compile test cases into binary code, launch VYPER with all checkers activated. These scripts also collect, normalize and synthesize the analysis results. The results are summarized in Fig. 6 and Table 2. We can notice that generally a good rate of false positives is obtained ($FP = 100 - TN$): from 0 to 2%, which is in correspondence to our objective to build a method with a very low false positive rate. Also, we remark that for the CWE 121 (stack overflow) we detected 0% exploitable vulnerabilities. This is due to the fact that the flawed code does not permit to completely erase the return address and cause an arbitrary code execution. In other words the CWE 121 test cases present in the Juliet test base are not exploitable. For the CWE 122, the low rate of detected vulnerabilities is caused by the fact that some heap overflows occur when reading heap buffers. The read access overflow is not considered exploitable in our definition because no heap metadata can be erased with such an access.

We equally tested several commercial and proprietary tools on the same test cases and we found that they have a mean value of false positive (FP) rate of about 20%. Hence, our tool performs well on the corresponding vulnerability classes. One explanation of such performance is that considered commercial tools are not tuned for exploitable vulnerabilities search and also they usually rely on techniques inherently generating a high rate of false positive alerts.

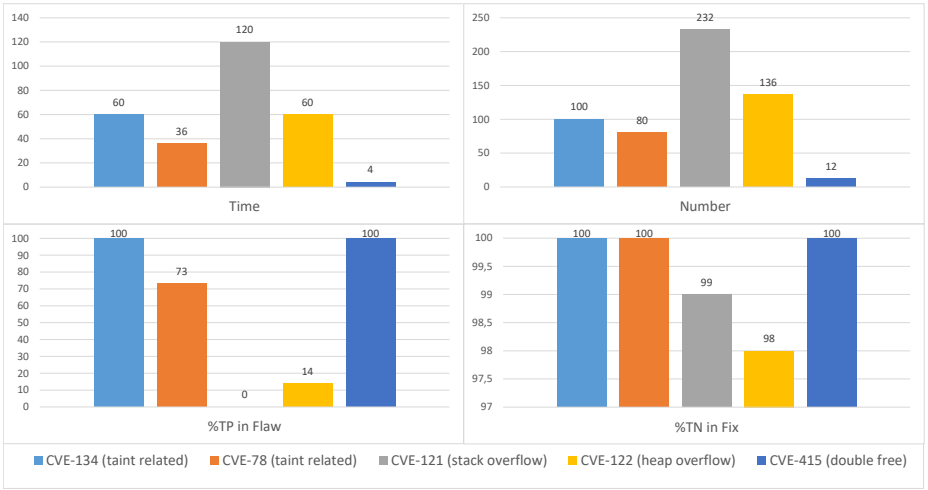

Fig. 6. Result of the analysis of a part of Juliet test suite. The charts present the analysis time, the number of test cases, the ratio of flaws correctly identified as vulnerable and the ratio of fixes correctly not identified as vulnerable.

Table 2. Result of the analysis of a part of Juliet test suite.

Type	CWE	Time[1]	Number[2]	%TP in Flaw[3]	%TN in Fix[4]
taint related	134	1h	100	100%	100%
taint related	78	36m	80	73%	100%
stack overflow	121	2h	232	0%	99%
heap overflow	122	1h	136	14%	98%
double free	415	4m	12	100%	100%

[1] total analysis time
[2] the number of analyzed test cases
[3] the ratio of flaws correctly identified as vulnerable
[4] the ratio of fixes correctly not identified as vulnerable

5.3 Testing on real applications

Testing our tool on real applications is an important step to show its effectiveness. Thousands of applications are now available in open-source. However, the intrinsic limitations of the *angr* framework allow us to perform the tests only on small or medium size applications. Another problem when dealing with open source code is how to state if the found vulnerability is correct or not automatically. We tested VYPER on the following of open source software:

- *Udhcp server*: udhcp-0.9.8 is a program running on a variety of devices (routers, modems, set-top boxes, IP cameras etc). We inserted into the source code a vulnerable printf call at udhcp-0.9.8/dhcpd.c:102. VYPER was able to correctly detect this vulnerability in about 3 minutes. This result shows the effectiveness of the tool for continuous testing of these types of (embedded) programs.
- *Widely used libraries*: We tested VYPER on OpenSSL-1.1.0f (libssl.so), libpng-1.5.20 and tiff-3.8.1. To test these libraries we launched the tool directly on the ".so" file and changed the entry point to each of the exported functions. We fixed the timeout to 300 seconds to be able to analyze a maximum number of functions in a reasonable time. The results are summarized in Fig. 7 and Table 3. As it can be seen several potential vulnerabilities were discovered.

6 Conclusion

In this study we have shown that using state-of-art binary-code analysis framework it is possible to effectively detect exploitable vulnerabilities. The method we propose tries to recognize the common patterns present in the application behavior allowing successful exploits. These patterns are searched using concolic execution engine provided by the *angr* framework. The implemented tool VYPER performs well on synthetic test cases as well as on real life applications.

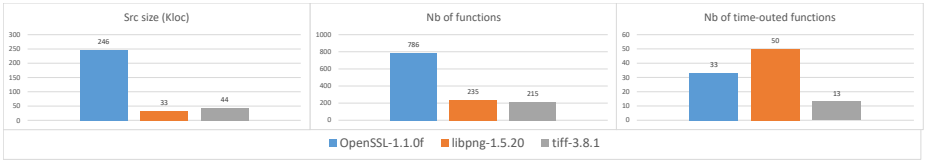

Fig. 7. Detection of vulnerabilities in widely used libraries. The charts present the binary size, the number of library functions and the number of functions on which the program reached a timeout.

Table 3. Detection of vulnerabilities in widely used libraries.

Library name	**Src size**	**Bin size**	**# f**[1]	**# tf**[2]	**Entry function:CWE id**
OpenSSL-1.1.0f (libssl)	246 Kloc	1.2 MB	786	33	SSL_add_dir_cert_subjects:CWE-122 SSL_check_private_key:CWE-121 SSL_CTX_set_ct_validation:CWE-121 SSL_CTX_use_PrivateKey:CWE-121
libpng-1.5.20	33 Kloc	452 KB	235	50	png_destroy_struct:CWE-415 png_do_unpack:CWE-122 png_free_default:CWE-415 png_info_init_3:CWE-415 png_push_process_row:CWE-121 png_safecat:CWE-121
tiff-3.8.1	44 Kloc	1.1 MB	215	13	TIFFCreateDirectory:CWE-415 TIFFCreateDirectory:CWE-122 TIFFGetConfiguredCODECs:CWE-122 TIFFInitCCITTFax3:CWE-122 TIFFReadEXIFDirectory:CWE-415 TIFFReadEXIFDirectory:CWE-121

[1] Number of functions
[2] Number of time-outed functions: the execution of the program reached a timeout without providing any results.

The results of running Vyper on these various test cases show its effectiveness and its ability to correctly detect and report exploitable vulnerabilities. The very low false positive rate is granted by the precise tracking of the vulnerable code behavior at binary level, where all the necessary details are available.

One of the major drawbacks of the developed method is its low speed with respect to other methods. Moreover, at the time of writing it does not scale very well for large applications.

The refine mode of VYPER can be used to check for potentially introduced vulnerabilities when application code is modified. This can be done by targeting the newly added lines of code and specifying the vulnerabilities to be checked.

In the future we plan to enhance the refine mode in order to have a better detection of newly added vulnerabilities. We also plan to concentrate on on the speed and the scalability of the tool in order to accept larger programs. Finally, we plan to explore other vulnerability patterns, in particular those related to race conditions and parallel executions.

References

1. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Commun. ACM **57**(2), 74–84 (Feb 2014). https://doi.org/10.1145/2560217.2560219, http://doi.acm.org/10.1145/2560217.2560219
2. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
3. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. pp. 380–394. SP '12, IEEE Computer Society, Washington, DC, USA (2012). https://doi.org/10.1109/SP.2012.31, http://dx.doi.org/10.1109/SP.2012.31
4. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. J. Comput. Syst. Sci. **7**(4), 354–375 (1973). https://doi.org/10.1016/S0022-0000(73)80029-7, https://doi.org/10.1016/S0022-0000(73)80029-7
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77). pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. SIGPLAN Not. **40**(6), 213–223 (Jun 2005). https://doi.org/10.1145/1064978.1065036, http://doi.acm.org/10.1145/1064978.1065036
7. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
8. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008)
9. Guide de règles et de recommandations relatives au développement d'applications de sécurité en java (2009), http://www.ssi.gouv.fr/uploads/IMG/pdf/JavaSec-Recommandations.pdf
10. Étude de la sécurité intrinsèque des langages fonctionnels (2011), http://www.ssi.gouv.fr/uploads/IMG/pdf/LaFoSec_-_Analyse_des_langages_OCaml_F_et_Scala.pdf
11. Common weakness enumeration CWE (2015), https://cwe.mitre.org/
12. angr framework (2016), https://github.com/angr/
13. CERT coding standards (2016), https://www.securecoding.cert.org
14. Coverity static analyzers (2016), https://www.coverity.com/
15. Exploit writing tutorial part 11 : Heap spraying demystified (2016), https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
16. HP fortify static code analyzer (2016), http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/
17. Juliet test suites (2016), https://samate.nist.gov/SRD/testsuite.php

18. Polyspace static analysis (2016), https://fr.mathworks.com/products/polyspace/
19. Smashing the stack for fun and profit (2016), http://insecure.org/stf/smashstack.html