Politecnico di Milano

Facoltà di Ingegneria
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea in
Computer Science

Binary Function Vulnerability Discover through LLVM IR

Relatore:
Stefano Zanero
Co-Relatori:
Mario Polino

Tesi di Laurea di:
Daniele Marsella
matr. 10482136

Anno Accademico 2019/2020

Abstract

The existence of security vulnerabilities in programs is one of the problems that mostly defines modern computer era; as a consequence, nowadays, software analysis has become one of the most relevant fields in computer security and an increasing number of researchers is currently working on the development of tools that could discover security vulnerabilities in the shortest time possible. In particular, there is a type of software analysis based on the analysis of program executables in binary format, which is binary analysis, that is quite useful because it can be employed to find vulnerabilities in softwares that have already been released.
Recently, the usage of *intermediate representation languages*, which are designed to simplify and enhance the analysis of program executables, has acquired an increasing interest in binary analyses researches. We decided to employ one of such languages also in our project in order to cover different machine architectures and also to make our tool easily extendable.

In our thesis we focus on a specific type of binary analysis, static binary analysis, which analyzes the executable without actually executing or emulating it, building an intermediate representation of it. We chose to use this approach because it ensures the coverage of the whole binary code inside the executable.

Our project's goal is to design a security tool that is capable of detecting a particular type of vulnerability, known as *buffer overflow*, in compiled softwares (binary executables). In particular, we are interested in a specific type of buffer overflow, the loop-based buffer overflow: this type of vulnerability happens when the program contains a loop that at each iteration stores an element of a source buffer into a destination buffer without checking the destination size. This loop is typically controlled by an user input, hence allowing the attacker to overwrite variables stored in program memory next to the destination buffer and, in the worse case, to execute malicious injected code. Moreover, this type of vulnerability is very common in strcpy-like functions.

In order to identify such vulnerabilities in binaries from different architectures, our tool translates the binary input into an intermediate language designed for program analyses and scans this intermediate representation to find any vulnerability. This solution allows also to produce a modular tool that can be easily extended and enhanced. After that, the tool scans the functions call chain of the program executable and tracks user input propagation from specific source functions to all the other functions, implementing

a simple taint analysis. This approach allows to filter out functions that contain buffer overflows but are not controlled by user input.

We designed three experiments to demonstrate the abilities of our tool in detecting buffer overflow vulnerabilities in different types of binaries. The first experiment tests tools abilities against both dynamically and statically linked binaries taken from public CVE lists of vulnerable programs. The second experiment tests tools abilities against binaries extracted from an ARM-based router's embedded firmware that had never been analyzed before. The last experiment tests our tools abilities against DARPA Cyber Grand Challenges example binaries, built on top of a custom operating system.

Our tests show that tool is able to identify 11 of the 15 vulnerabilities chosen in the first experiment, as well as a not yet discovered vulnerability inside one of the binaries of the firmware for the second experiment. The results of the third experiment show that the tool marks as vulnerable at least one function in all the binaries that are known to contain a buffer overflow vulnerability. Overall results demonstrate that our tool can be efficiently used to simplify vulnerability detection in binaries, but it still requires improvements on the detection precision.

Sommario

L'esistenza di vulnerabilità di sicurezza nei programmi è una delle problematiche che maggiormente caratterizzano la moderna era informatica, tanto che ormai l'analisi del software è divenuta uno degli argomenti più discussi e un sempre crescente numero di ricercatori in una corsa contro il tempo alla ricerca di una soluzione efficace che permetta di individuare le vulnerabilità nei programmi il più velocemente possibile. In particolare, esiste una specifica tipologia di analisi dei programmi basata sull'analisi degli eseguibili dei programmi in formato binario, chiamata binary analysis, la quale risulta particolarmente utile perché può essere utilizzata per trovare vulnerabilità in software che sono già stati rilasciati.
Di recente l'utilizzo di *intermediate representation languages*, che sono linguaggi ideati per semplificare e migliorare le analisi di software eseguibili, ha ricevuto una particolare attenzione nelle ricerche di binary analysis. Per questo motivo, anche noi abbiamo deciso di adottare uno di questi linguaggi all'interno del nostro software, in modo da poter gestire facilmente diverse architetture e rendere il nostro tool modulare.

Nel nostro progetto ci soffermiamo principalmente sulla static binary analysis poiché essa garantisce la copertura completa del codice all'interno del binario, mentre la dynamic binary analysis non può testare tutti gli input del programma per via delle dimensioni in genere troppo grandi del dominio degli input.

In questo progetto presentiamo un tool di sicurezza informatica in grado di trovare un particolare tipo di vulnerabilità software, chiamato *buffer overflow*, all'interno di programmi già compilati. In particolare, il nostro progetto si sofferma su un particoalare tipo di buffer overflow, i loop-based buffer overflow: questo tipo di vulnerabilità è presente nei programmi che contengono loop che ad ogni iterazione copiano un elemento da un buffer sorgente a un buffer destinazione, senza controllare le dimensioni della destinazione. Generalmente, la condizione di tali loop è controllata dall'input dell'utente e questo permette a un possibile attacker di poter sovrascrivere varibili in memoria adiacenti al buffer di destinazione, permettondogli inoltre, nel caso peggiore, di eseguire del codice malevolo.

Il nostro tool traduce i binary in input in una rappresentazione intermedia utilizzando un intermediate representation language, dopodiché analizza tale rappresentazione in cerca di vulnerabilità. Questa soluzione ci ha permesso di creare un tool modulare che può essere facilmente esteso e migliorato. Inoltre, il tool è in grado di tracciare la propagazione dell'input dell'utente,

attraverso le chiamate di funzioni, a partire da specifiche funzioni sorgente. Questo permette al nostro tool di eliminare dai risultati quelle funzioni che contengono dei buffer overflow che non sono controllati dall'utente.

Abbiamo delineato tre esperimenti che dimostrano le abilità del nostro tool nella individuazione delle vulnerabilità di tipo buffer overflow all'interno di binari di diverso genere. Il primo esperimento testa le abilità del tool usando binari linkati sia dinamicamente che staticamente a librerie esterne, presi da liste pubbliche di programmi vulnerabili. Il secondo esperimento, invece, testa le sue abilità usando binari estratti dal firmware di un router basato sull'architettura ARM. Il terzo e ultimo esperimento infine testa il tool impiegando i binari di esempio offerti dalla DARPA Cyber Grand Challenge, costruiti usando un particolare tipo di sistema operativo usato per queste competizioni.

I nostri test dimostrano che il tool è capace di identificare 11 dei 15 binari pubblicamente noti come vulnerabili nel primo esperimento, e ci ha permesso inoltre di identificare una vulnerabilità non ancora nota all'interno di uno dei binari estratti dal firmware del secondo esperimento. I risultati del terzo esperimento mostrano inoltre che il tool è stato capace di identificare come vulnerabili tutti i binari contenenti almeno un buffer overflow. Nel complesso, i risultati dimostrano che il nostro tool può essere utilizzato efficacemente per semplificare l'individuazione di questo tipo di vulnerabilità, anche se necessita ancora dei miglioramenti per la precisione dei risultati.

Contents

List of Figures

List of Tables

1 Introduction

The production of secure softwares has become of the utmost importance both for developers and businesses that are based on software technologies, though detecting vulnerabilities in a program is not always a straightforward task.

In this thesis we are going to present a security tool, developed using modern compilation and computer security technologies, capable of detecting security vulnerabilities in already compiled softwares. The solution proposed by our project is able to identify vulnerabilities at function-level inside binary executables, already deployed and used in different use-cases, extracting a lot of information about the usage of function arguments and dumping them into a standard format that can be further analyzed later by other tools.

Proprietary softwares are shipped with only the executable binary, so users cannot analyze their source code for vulnerability. This is the case also for the so called Component Off-the-Shelf or COTS software, and the same reasoning applies also to legacy softwares, which are old programs critical in particular businesses that cannot be replaced and whose source code in many cases is not available, or even embedded softwares, used in many electronic devices like routers or modems . In all these cases, the only way to discover vulnerabilities is through a direct analysis of binaries.

A common example of vulnerabilities discovered in binaries is buffer overflow,which is a type of vulnerability among all type of applications actually present on the market. A buffer overflow occurs when a value is stored beyond the memory region occupied by the variable that should contain that value. The most common example of this kind of vulnerability is the *strcpy* function in C standard libraries: this function copies the characters from a string and transfers them to another one, but in many implementations the destination buffer size is not checked, causing some characters to override values of contiguous variables in memory. The buffer overflow vulnerability is often used by many attackers to inject vulnerable code inside variables and to manipulate the program instruction order to execute it.

Over the years, many countermeasures have been developed to secure programs against buffer overflow, but attackers have always managed to find a way to elude them. One of the first countermeasure consisted in a canary before the return address stored on the stack, which was introduced by compilers in order to detect if there was a buffer overflow during execution and then make the program quit instantly. Unfortunately, the first implementations of this solution used a fixed canary that did not change between different invocations of the program, and which made it possible for attackers to overwrite the canary with the same value, consenting them to bypass the whole countermeasure.

Therefore, compilers developers began to work on an enhanced version of this countermeasure, which required the canary to be generated randomly at each function invocation: theoretically this manoeuvre should have made it impossible for malicious actors to know canary value before program execution. However, even in this case, these actors found a way to reconstruct the value of the canary, typically by exploiting some information leakage of the application.

Years later, even some of the most important computer companies started to study and develop their own version of a countermeasure which could be able to avoid buffer overflows attacks, one of the most prominent was the GCC StackGuard detector proposed at USENIX Security Symposium in 1998 [6]. But still attackers were able to counteract to this new solution thanks to other types of overflows, which were not covered by the newly released countermeasures, like heap-based buffer overflow.
A review of the known attacks and possible defenses was in a paper by C.Cowan et al. [5] published at the beginning of 2000.

Nowadays it is still impossible to write a perfectly secure software against buffer overflows because of the many possibilities that attackers discover, which enable them to exploit program vulnerabilities. Every year many computer security competitions are organized to search for all possible attacks on binaries, hoping this would help developers to figure out new defenses for their applications.

For these reasons, and also since buffer overflow is still one of the most common vulnerability (the CVE database CVE Details show that only in 2019 1,247 overflow vulnerabilities were discovered, making it one of the most common types among all 2019 vulnerabilities), we focused our effort on the detection of this particular type of vulnerability.

We have tested our tool using different binaries extracted from different use cases, analyzing the performance of the results and designing future solutions that can be employed to enhance detection precision and also add new functionalities to the tool. With our tests we have been able also to identify a new buffer overflow in a firmware installed on a ARM-based router, vulnerability that was never detected before.

2 Background and Motivation

2.1 Binary Analysis

Binary analysis is the technique used to scan programs executables (that are typically in binary format) for different reasons, first of all to find vulnerabilities and bugs directly inside it. While different binary formats provide features that simplify many binary scan operations (i.e., symbol tables and sections), it is much more difficult to analyze binaries behavior with respect to source code because it contains much less information. For, example in machine code all values are interpreted as integers and there is a finite set of registers used to access all program variables, one at a time, while many high level languages allows programmers to declare variable types and to use an infinite number of variables. Moreover, machine code expressiveness is much lower than the one of any high level language, and for this reason there is a loss of information during compilation process. The information removed by compilation often is about non-executable aspects of the program and exposes information about its semantic that is very useful for security analyses, while any vulnerabilities discovery tool strives to reconstruct the same information from only machine code. Lastly, another disadvantage is introduced by aggressive compilation optimizations that alter an intermediate representation of the source code before producing the final binary executable.

In the last decade, many researchers developed numerous projects in binary analysis, steadily improving this field of computer science.

Modern binary analyses are based on *control-flow* and *data-flow* inspection of programs. Each of them is a different application of the concept of *flow-graph* applied to a different property of the binary.

2.1.1 Control-Flow Graph

A Control-Flow Graph (CFG) is a representation of the instructions and branches contained in a program and order in which they are executed. Typically each node in a control flow represent a single instruction (or a sub-set of instructions without branches) and each of them can be connected by arrows to one or more node of the diagram. Nodes with outgoing links to more than one instruction are called *branch instructions* and contain conditions which are useful to choose which of the following instructions will be executed. An example of a simple CFG is represented in Figure 2.1 : I1, I2,. . . , I7 are simple instructions and C1, and C2 are branch instructions.

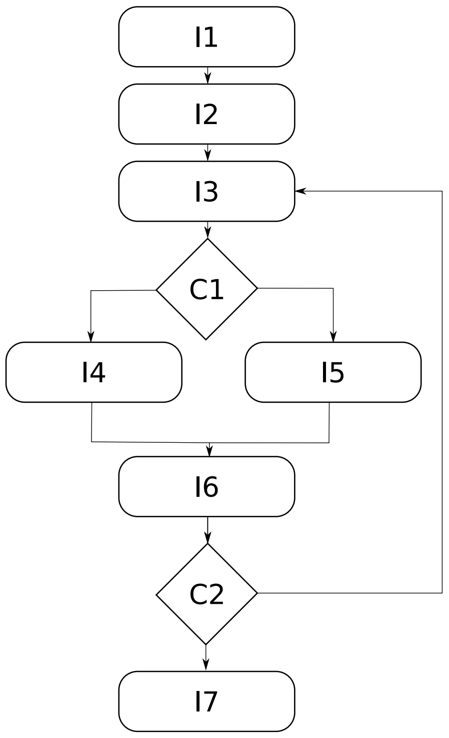

Figure 2.1: An example of a Control-Flow Graph

For a binary analysis it is important to build or recover a CFG of the input binary in order to have a correct representation of the possible outcome of the program execution. Branch instructions conditions are analyzed by tools that help understanding when it is possible that an instruction or control-flow path is taken or, more importantly, they help identifying *dead instructions*, that are instructions which are never executed, since they include a condition a condition that can never be true.

2.1.2 Data-Flow Graph

Another important property of binaries for analyses is the *data-flow.* While control-flow takes into account how instructions are related during execution, data-flow keeps track of values flows between variables. A *data-flow graph* or DFG contains different types of nodes: a *variable node* defines where and how a variable is defined (the incoming edge indicates how it is defined and the outgoing one how it is used), an *operational node* defines operation that can transform variables values . Figure 2.2 shows an example of what a DFG looks like.

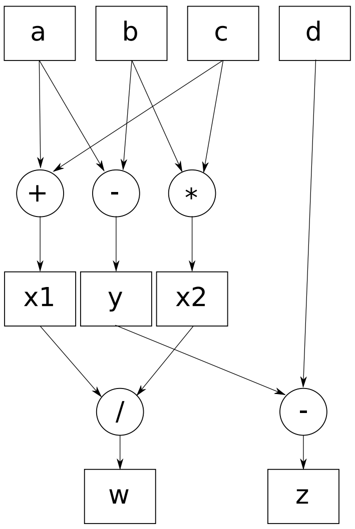

Figure 2.2: An example of a Data-Flow Graph

DFG are very useful in binary analyses: they can be exploited to recover value ranges for variables and for pointer-aliases analyses too. Pointer-aliases analyses are also very important in control-flow reconstruction, because they can help finding the targets of indirect jumps and calls. Data-Flow analyses are able to identify malicious assignments on variables, or bad behavior in the program that can lead attackers to execute malicious code, like the already mentioned buffer overflow exploit.

2.2 Static vs. Dynamic Analysis

Nowadays binary analysis is grouped into two macro-areas based on two different approaches for vulnerability detection in already compiled programs, and each of them has its own pros and cons:

- Dynamic Binary Analysis
- Static Binary Analysis

2.2.1 Static binary analysis

Static binary analysis is the technique to scan input binary without executing or emulating it, but by merely reading the machine code contained in the input. This kind of binary analysis aims at understanding the assembly code, reconstructing which function calls other functions and which memory area is pointed by any pointer in each execution time. In order to work properly, static binary analysis often needs to draw some assumptions on input binaries and make massive use of different advanced mathematical tools. Anyway, even with these measures, sometimes it still is not able to obtain precise results, thus demanding the use of different statistical tools to make predictions on the hypothetical execution of the binary, i.e., which branch is taken at a given point by a conditional instruction. To sum up, the main challenges in static binary analysis are:

- Value ranges for variables
- Memory Areas pointed by different pointers (also known as pointer aliasing)
- Branch conditions predictions

Clearly though, the main complication of this kind of binary analysis consists in the difficulty in reasoning about low-level code because of its nature, leading to the recent birth of different *intermediate languages* that try to make it Static Binary Analysis easier. A more in depth view of the most used intermediate languages will be covered in a later chapter.

Finally, one of the biggest project on static binary analysis in recent years has been BitBlaze [22], that is a complete binary analysis framework for computer security, with a component called Vine studied specifically for static analyses of binaries.

Intermediate Languages

For many years static analysis of binary was built on top of reverse engineering, an approach that tries to translate back binary into its source code. The main idea behind this approach was to analyze the recovered source code for vulnerabilities, but it proved to be quite ineffective in practice: due to compiler optimizations, the recovered source code turned out far different from the original code and in many cases was very difficult to understand by analyzers. In an attempt to resolve these complications, researchers started to design languages that had binary analysis as the main goal. The idea of an intermediate language between the high level code and the machine code was not new, in fact compilers already used intermediate languages to simplify code optimization process, but those intermediate languages were not well suited for computer security analyses. This is why new intermediate representation languages (IL) were born, like Vine, the Bitblaze intermediate language or BIL for Binary Analysis Platform [3]. The latter is a type of IL that represents the operational semantics of the binary input on which all analyses will depend, implying also that any error or bug in the IL might invalidate all previous analyses and emphasizing the critical role of IL design. Furthermore, the translation to IL of a binary, a process also called *lifting*, starts from an input language that is incredibly complex: ARM and Intel specifications of processors machine code are reported in manuals of 6,354 and 4,700 respectively. The research of Kim, Soomin et al. in 2017 [10] tested recent ILs of the most known binary analysis tools looking for semantic bugs. The research tested the expressiveness of each intermediate language by looking at two important characteristics of any IL: *explicitness* and *self-containment*. An IL is explicit when no instruction updates more than one variable, a property very useful for control-flow and data-flow based analyses. On the other hand IL is self-contained when it represents all the binary characteristics without relying on external functions or components. It also means that the IR is *side-effects free*. For example QEMU IL, called TCG, uses some external functions for some logical operations, and for this reason it is not self-contained. While self-containment is very important for the expressiveness of the IL, a non-explicit IL is not always less expressive than an explicit one. Thanks to their works, the researchers found 23 semantic bugs in three state-of the art lifters, which helped us choosing a lifter based on `LLVM` IR.

2.2.2 Dynamic binary analysis

Dynamic binary analysis is based on the execution or emulation of the input binary inside an *instrumented environment*, that allows to test program with real inputs, by not only supporting the simulation of execution, but also consenting its modification and without side-effects on the outside environment. It is an efficient approach in terms of response time and complexity of design, but it does not ensure that the whole code is covered. The main problem of this type of analysis is that the input domain can be very large even for a simple program, thus making it essentially impossible to cover the whole domain by enumerating of all possible value, even with the help of most powerful supercomputers.

It is precisely for this reason that researchers developed some tools that try to cover as much input domain as possible, drawing information from different approaches. Nowadays, the most common dynamic analyses are based on *input fuzzing*, a technique that scans the program for invalid input which could make it crash or behave unexpectedly. Since these input data can be generated randomly or algorithmically, one of the most common solution is to generate input that tries to cover all possible conditions for branches instructions.
A study conducted by Shoshitaishvili et al. [21] on binary offenses identifies two common types of fuzzing used in dynamic analyses: *coverage-based fuzzing* and *taint-based fuzzing.* Coverage-based fuzzing tries to find input that maximize the coverage of executed code. The idea behind this approach is that executing most instructions in the code raises the probability of discovering an hidden vulnerability in the program. While this can be true in many cases, it is also possible that an instruction leverages a vulnerability only when it is executed with some specific inputs, so this approach does not ensure that all vulnerabilities are discovered. This type of approach is used in a famous state of the art fuzzer, named American Fuzzy Lop or AFL [25]. Taint-based fuzzers instead track how the input is propagated inside the program (typically observing the function call stack) and use this information to modify input that are generated next. These fuzzers are typically useful when the analysis has a specific target function to track, but in a more general approach it is very difficult to understand how to modify input in order to identify an hidden vulnerability. Taint-based fuzzers are still under development, but one of the most promising is the one presented in the paper by S. Bekrar et al. [2] Another famous fuzzer, maybe the most powerful fuzzer actually available, is OSS-Fuzz [19], developed by Google : it is able to generate 4 trillion inputs a week, and it is mainly used to search bugs in Google's software, like Google Chrome. This fuzzer is so effective that, until now, it has been able to find over 16,000 vulnerabilities in more than 250 open-source projects

The paper by Shoshitaishvili et al. [21] also identifies another more complex and recent approach used in dynamic binary analysis, the *dynamic symbolic execution.* This technique represents a mix of static and dynamic approaches: first the binary is statically analyzed to identify and understand expressions and conditions inside of it; then a symbolic representation of each of them is built, highlighting how input can change their values; in the end these symbols are used as insights by a fuzzer for input generation. Dynamic symbolic execution has the added value of a more concrete knowledge of the program semantic compared to to classical fuzzers, allowing the analyzer to target specific states of the program by simply looking at conditions or expressions and propagating its symbols backwards to the input. This powerful technique can be used for different purposes: classical dynamic symbolic execution engines employ it to directly find vulnerabilities inside the program

Dynamic binary analysis is sometimes also used to validate results of static binary analysis, typically to narrow down the number of false positives reported.

2.3 State of the Art

As of today, the most popular static binary analyzers are BAP and BitBlaze, while for the decompilation of binaries business tools like IDA Pro and open-source projects like radare2 are commonly used.

On the subject of reverse engineering, there are two main open-source projects that use `LLVM` IR as intermediate representation language (IR): Remill and Retdec. Retdec was developed by Avast [11] and is the most recent reverse engineering project, it already offers many advanced functionalities like RTTI and Class Hierarchy reconstruction for C++ projects, instruction idioms recognition and stack structure reconstruction. Actually Retdec has been opened to the public and its source code is accessible on Github, and, even if it is still an ongoing project, its performances are already incredible: it is able to lift a complex binary of some megabytes in a couple of minutes at most.

Unfortunately though, the decompilation feature to C language is not perfect yet and the output code can still come out as “dirty”, with a lot of goto instructions, making it a bit more difficult to analyze compared to standard C code.

2.3.1 BAP: Binary Analysis Platform

BAP is the third version of the project developed by David Brumley et al. [3] to build a unified platform for binary analysis. The first version of the project started with a simple decompiler, called asm2c, that translated binary code into C code and then performed analyses on the high-level language. Then developers opted for a custom IR, called *Vine*, that was based on VEX, the IR developed and used by Valgrind [15] (another state of the art project that will be discussed later). In this last version researchers tried to fix Vine in order to make all code side-effects explicit in the IR, allowing to perform syntax-based analyses. BAP is now formed by two components: a front-end that takes the binary as input, search for executable code in it and translate it into Vine code, and a back-end that performs analyses based on

Vine language. Additionally, the back-end can be interfaced with other tools to enrich analyses results (i.e., it can be interfaced with an SMT solver to compute better data-flows or with Intel Pin framework for dynamic symbolic execution). Another feature introduced with the last version of BAP are the *Verification Conditions* (or VC): BAP can analyze the program and build particular conditions that can be verified and tested with specific inputs over all program execution.

Currently BAP supports only binary compiled for x86 and ARM architectures.

2.3.2 Valgrind

While BAP represents the state of the art of static binary analysis, the project developed by Nethercote et al. [15], called Valgrind, represents the actual state of the art for dynamic binary analysis. Valgrind is a dynamic-binary instrumentation framework designed to build heavyweight dynamic-binary analyses and shadow values tools. Shadow values tools are particular dynamic-binary analysis that, during program execution, replace each emulated register value with a description of that value, allowing the recovery of a brief history of that value for each register. In addition to simple instrumentation framework, shadow value tools have additional complex requirements that must be met for their building. Valgrind paper defines nine requirements that are grouped into four categories.

A shadow tools has to keep a program state that represents memory areas used by variables. Then, each possible machine instruction must be instrumented in order to populate descriptive values about each shadow value.

All analyses and interpretations are performed using an intermediate representation language developed by Valgrind researchers: this intermediate representation language, called VEX, had received a lot of changes until the third implementation, when it was finally reliable enough to be used as default IR. VEX language is an architecture independent, SSA language, similar to RISC machine languages composed by blocks called *superblocks*. Each superblock contains a list of instructions and each instruction is composed by one or more expressions depending on the behavior of the instruction. Instruction expressions can be represented in two ways: a tree VEX IR contains expressions arranged in tree-form, or a flat VEX IR that contains only linear expressions. It is always possible to convert a tree VEX IR into a flat VEX IR, and viceversa.

Valgrind is composed by a core program and additional plug-ins that are attached to the core when the tool starts. With this architecture, Valgrind users can easily implement new tools by writing only new plug-ins, reusing the same core features. The core is responsible for translation into the VEX language using the *disassemble-and-resynthetize* (D&R) approach, in contrast with more common *copy-and-annotate* (C&A) approaches used by other dynamic instrumentation frameworks.

Thanks to its ability, Valgrind is actually used in different common tools employed in many computer security fields: *Memcheck*, developed by Seward Julian and Nethercote Nicholas [20], is a tool capable of identifying frequently undefined errors vulnerabilities, common in programs written with unsafe imperative languages like C or Fortran; *TaintCheck*, developed by Newsome James and Song Dawn Xiaodong [16], is a dynamic taint analysis tool able to follow input propagation from specific source functions in instrumented binaries, automatically detecting input correlation of exploit attacks; *Redux*, developed by Nethercote Nicholas and Mycroft Alan [14], is a dynamic data-flow tracer tool able to build dynamic data-flow graphs (or DDFG) in analyzed binaries, that represents the entire computation history of the program. Redux, in particular, developed a whole new way to analyze program using DDFG, allowing also to understand how values can affect the outside environment.

These are only few of the many use-cases in which Valgrind has been used to develop dynamic binary analysis tools.

2.3.3 BitBlaze

BitBlaze is a project conducted by Dawn Sang et al. [22] that aims at providing basic and common useful tools for binary analyses in computer security, and from those it builds new complex and effective solutions to security problems.

It was designed with accuracy and extensibility in mind, building models that can accurately represent program execution and re-using core functionalities for more sophisticated and complex analyses. BitBlaze developers decided also to mix together static and dynamic binary analyses to benefit from both advantages for their results.

To achieve that, BitBlaze relies on three components: *Vine* component for static analyses tasks, *TEMU* component for dynamic analyses tasks and *Rudder* component for mixed concrete and symbolic analyses based on the other two components. Figures 2.3, 2.5, and 2.4 from the research paper offer an overview of these Bitblaze components.

Rudder is the most innovative part of the project: it takes as input symbolic specification computed by TEMU and explores execution path condi-

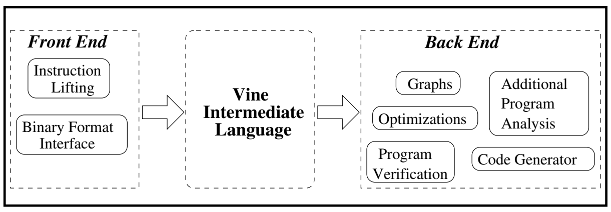

Figure 2.3: An overview of Bitblaze Vine component

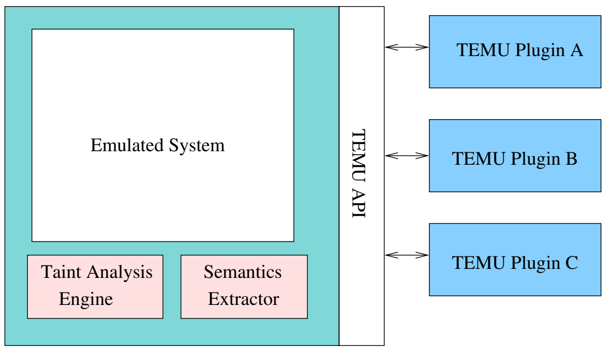

Figure 2.4: An overview of Bitblaze TEMU component

tions in the binary, automatically discovering behaviors hidden by particular complex conditions. The *Mixed Execution Engine* (MEE) behind Rudder is able to collect necessary information from symbolic execution and it is also able to formulate a *symbolic program* of the binary: in fact when a symbolic execution marks an instruction to be executed, it is first lifted to the Vine IL, allowing to reconstruct a symbolic version of the program in the end . In addition to it, MEE uses also an SMT solver to reason about symbolic expressions provided by TEMU.

2.4 LLVM : A MODULAR COMPILER FRAMEWORK

Low Level Virtual Machine (LLVM) framework was born from the effort of researchers C. Lattner and V. Adve [12] who wanted to design retargatable compilers that, starting from the same code, were able to produce executables for different architectures. LLVM 's most valuable feature is that it is able to easily introduce new optimizations in the compiler without specific knowledge, neither of the input high level language nor of the target architectures. This feature was achieved thanks to the well-engineered design of the

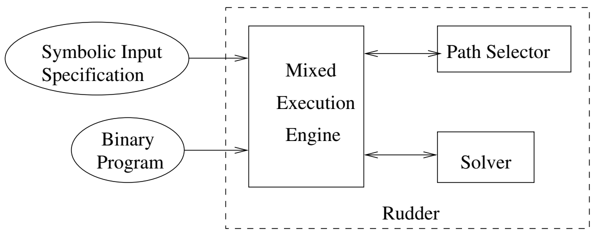

Figure 2.5: An overview of Bitblaze Rudder component

framework and also through the effort put in the implementation of the Intermediate Representation Language `LLVM` IR. In less than ten years `LLVM` has managed to become the main competitor of the old and great GCC compiler and it has developed so much that now Clang, that is C and C++ compiler based on `LLVM` , is the default compiler used by Apple in their operating systems .

2.4.1 `LLVM` IR

The core strength of the `LLVM` framework is its well engineered *intermediate representation language* named `LLVM` IR: it is used to translate any input high level language into a middle form that can be easily manipulated and optimized independently from the target architecture and the input language. The Listing 2.1 contains an example of a `LLVM` IR sample program.

```
@G = thread_local(initialexec) global i32 0, align 4
%mytype = type { %mytype*, i32 }

declare i32 @printf(i8* noalias nocapture, ...)
declare i32 @atoi(i8 zeroext)
declare signext i8 @returns_signed_char()

define i32 @main() #0 {
  %1 = alloca i32, align 4
for.body:
  %val0 = load i32, i32* %1, !llvm.access.group !1
  store i32 %val0, i32* %1, !llvm.access.group !1
  br i1 %exitcond, label %for.end, label %for.body, !llvm.
     loop !0
for.end:
  ret i32 0
}
```

```
; Some unnamed metadata nodes, which are referenced by the
    named metadata.
!0 = !{!"zero"}
!1 = !{!"one"}
!2 = !{!"two"}
; A named metadata.
!name = !{!0, !1, !2}
```

Listing 2.1: An example of LLVM IR Module

LLVM IR's structure is very similar to assembly language, with three-addresses statements and not so many high level constructs, but at the same time introducing some important features that increase its expressiveness: it has a full type system for variables, it allows the definitions of functions and local or global variables and, most importantly, it is in Static Single Assignment form (SSA).

Definition. *A language is in SSA form when any modification or use of variables leads to the definition of a new variable.*

This property makes it very easy to decouple different uses of the same variable in different statements of the program. This also means that theoretically it is possible know to which value a variable will assume for every single statement in any execution moment (or at least a range of possible values). The use of this structure also facilitates tracking the flow of a variable with the so called *define-use* chains (or def-use chain): in this type of chain any ring is composed by an use of a variable previously defined in the program and a definition of a new variable. If two rings are consecutive, the second ring must contain a use of the variable defined in the first one. Formally a def-use chain can be defined as following:

Definition. *A define-use chain in an SSA form language is a ordered sequence of instructions $< i_1, \ldots, i_n >$ where the following property must hold:*

$$\textit{if } < i, j > \textit{is a sub-sequence of the def-use chain} \Rightarrow \exists v \textit{ s.t.} \begin{cases} v \in def(i) \\ v \in use(j) \end{cases}$$

where $def(i)$ is the set of variables defined by i and $use(j)$ is the set of variables used by j.

SSA form languages are preferred for data-flow analysis, in fact thanks to SSA property each variable in the DFG has a single definition node.

The structure of a single LLVM IR compilation unit, also called *Module*, is composed by blocks: the top-most block is the *Module* block that contains all other blocks in the module. Inside a Module blocks there are different

Function blocks, each of them representing a single function in the program. One Function block can be formed by only *Basic-block* blocks or also by *Loop* blocks (containing only *Basic-blocks*). A Basic-block is a block of instructions that are executed always in the same order and it is defined typically by a *label* at the beginning and a ending instruction (called also *Terminator Instruction*) that connects it to other blocks. This type of block is the unit that compose the CFG of the `LLVM` IR Module. An high-level overview of a generic `LLVM` IR Module structure is represented in the Figure 2.6

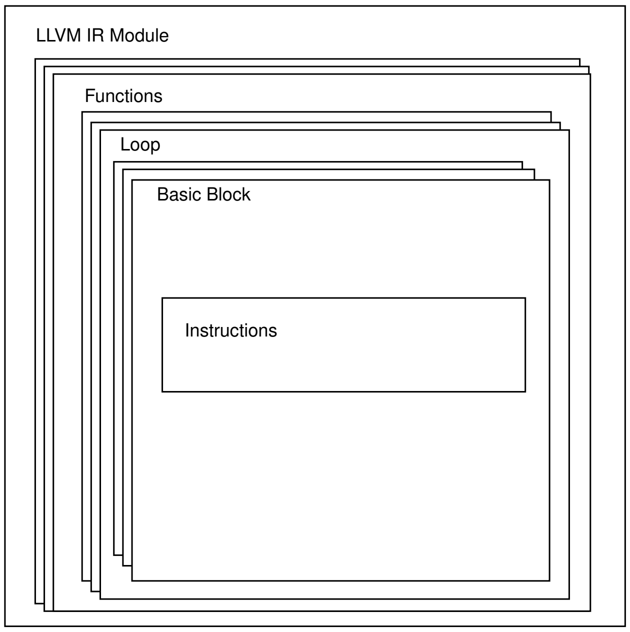

Figure 2.6: A representation of the structure a `LLVM` IR Module

Inside the Basic-blocks there are only Instruction blocks, each of them representing a single `LLVM` IR Instruction. Additionally, a Module can also be enriched with other information about the program using *metadatas*: a metadata is a non-executable instruction or variable that can be attached to each component of the module, adding additional information that can be useful for optimizations or debugging.

The most relevant aspect for developers using `LLVM` Framework is that each element of the IR Language is implemented in a beautiful and well-engineered hierarchy of C++ classes in the `LLVM` libraries. The hierarchy starts with the top most class, that is *Value*: anything inside the `LLVM` IR is a direct or indirect subclass of it. At the same time, any value inside the IR can be an User of other values and the relation between user and used values is represented by Use class. Then, using the polymorphism properties of C++, all other types and constructs of IR are implemented. Some of them are:

- Constants
- Instructions
- Basic Blocks
- Functions
- Loops

Each of them inherits or overrides *Value*'s methods depending on the behavior. All the objects in the libraries are implemented following at least C++ 11 standards, making it very easy for developers to iterate over a vector of variables uses, for example using standard methods ".begin()" and ".end()" or the most common standard libriaries of C++. Anyway, in order to enhance performance, `LLVM` designers decided also to reimplement some of the basic types in C++, like arrays with ArrayRef or SmallVector and strings with StringRef and Twine. Third party developers have the possibility to continue using standard C++, but if they need to increase the performance of their tools they have to take into account also `LLVM` base classes. All these aspects are well documented inside the `LLVM` programmer manual which lists many examples that help developers with design decisions.

LLVM Passes

All optimizations and transformations performed by `LLVM` are built on top of the concept of Pass, that is a component which has to perform a specific operation on the input module at a specific granularity. Passes can be categorized into two classes:

- Analysis Pass: a pass that reads the input module and does not modify it, retrieving information that could be useful for other passes.
- Transformation pass: a pass that actually modifies the input producing a new module. It can invalidate previous analyses.

Passes are used by `LLVM` optimizer tool, called *opt*, that builds a pass pipeline in which passes are ordered based on their dependencies on other passes. Each pass can have different granularity, that specifies on which IR level the pass is iterated:

- *Module pass*: a pass that iterates once on the whole input module having the possibility to modify and analyze any component.
- *CallGraphSCC pass*: a pass that iterates over the module call-graph, letting the developer decide in which order to visit it (depth-first or breadth-first).
- *Function pass*: a pass that iterates on each function of the input module and that can modify a function and all its internal component one at a time.
- *Basic Block pass*: a pass that iterates over the basic blocks of the module and can modify only instructions inside of it.
- *Loop Pass*: a pass that iterates over all the loops inside the module and that can modify only the instructions inside it
- *Instruction Pass*: the most granular type of pass, it iterates over all instructions of the module and it can modify one at a time.

An important note must be discussed about CallGraphSCC passes: since a CallGraph can contain loops caused by recursive functions or groups of functions calling each other, it is transformed into an acyclic graph before execution using *Strongly Connected Component* (or SCC) algorithms. The default algorithm used by `LLVM` is the Tarjan's SCC algorithm, defined by Robert Tarjan in its study about depth-first visiting of linear graphs [24]. Eventually, developers can modify its behavior by overriding the default pass constructor. Then, developers have to override the "runOnSCC" method, that analyzes one SCC of the CallGraph at a time. Each SCC is formed by one or more functions, with relation links that represent which function calls other functions inside the SCC. Finally, developers are able to analyze the whole CallGraph overriding the method "doFinalization" that takes as input the CallGraph without SCCs.

`LLVM` Pass Managers

The passes pipeline in `LLVM` is built by a specific class called PassBuilder. After building, another class (the Pass Manager) executes each pass in order and registers which analyses have been performed and which need to be re-executed. This Pass Manager supports the class hierarchy of passes described in the previous section. Each pass can declare, through an overloaded function, on which analyses and transformation it depends on; after that, PassBuilder and PassManager can ensure that they are executed in the correct order. In the `LLVM` documentation it is assured that each pass can depend only on specific types of other passes, which are:

- Module Pass can depend on any type of pass.
- CallGraphSCCPass can depend only on other CallGraphSCCPass.
- Function Pass can depend on other function passes and basic block, loop pass or instruction pass.
- Loop pass can depend on other loop passes and basic block or instruction pass.
- Basic Block Passes can depend on other basic block passes or Instruction passes.
- Instruction Pass can depend only on other Instruction passes.

Unfortunately, the Pass Manager is affected by a bug due to its design, which invalidates the dependencies of passes on other types of passes: the problem is caused by the instantiation of each Pass inside the Pass Manager, which leads to a crash of the program when a pass tries to access a result for a unit from different the one that is being analyzed in that moment from that instance.

New Pass Manager The above-mentioned problem, alongside other known disadvantages of the current implementation, lead `LLVM` developers to work on a new pass manager restructuring all the previous designs. With the new pass manager, passes are classified into two different class hierarchies: *AnalysisInfoMixin* and *PassInfoMixin*. In this way it is easier to identify which passes can modify the input module. Additionally, the passes do not declare if they modified the module with a simple boolean variable, but they return a set of Preserved analyses and transformations at the end of their execution. In this way the pass manager can reschedule only needed analyses and not all of them, while the old pass manager re-executed all previous analyses. Another important change has been introduced in the new pass manager to simplify the access to analysis results: each analysis can declare a specific internal type named "Result" that must be returned at the end of the analysis. The new pass manager registers all analyses in the pipeline and cache also results for already processed units. During the execution each pass can access the instance of the pass manager and ask for a particular result of analyses. If the analyses has a different granularity, the pass can access a particular proxy built by pass manager to facilitate the communication among those types of passes.

At this moment the new pass manager is still in beta state, but since `LLVM` 7.0.0 both versions coexist and developers can choose which type of pass manager to use for their analyses.Moreover, the new pass manager still has some disadvantages (no possibility to define pass-specific command line options for example) so we had to use the old pass manager approach for our project.

2.5 `REV.NG` : A UNIFIED BINARY ANALYSIS FRAMEWORK

`REV.NG` [8] is a unified binary analysis framework covering a very large set of processor architectures, and at the same time it is also a decompilation and translation tool that can translate a binary compiled for a specific architecture into an executable binary for a different architecture. Differently from BAP , `REV.NG` is able to target so many architectures because it relies on QEMU (a generic open-source machine emulator and virtualizer) as front-end for input binaries and it uses Low Level Virtual Machine IR (`LLVM` IR), which is architecture-independent by design, as intermediate representation language for the analyses. Thanks to this decision, `REV.NG` has the ability to manipulate binary from different architectures without having different specific components for each of them (while other common binary analysis tools have to). The Figure 2.7 from the original paper shows an overview of the system behind `REV.NG` .

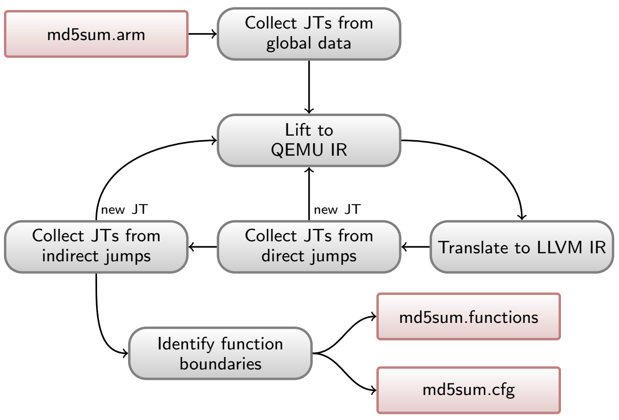
md5sum.arm
Collect JTs from global data
Lift to QEMU IR
new JT
new JT
Collect JTs from indirect jumps
Collect JTs from direct jumps
Translate to LLVM IR
md5sum.functions
Identify function boundaries
md5sum.cfg

Figure 2.7: An overview of the `REV.NG` system. JT stands for *jump tables*, and new JT means that a new *jump target* has been found

2.5.1 CFG reconstruction

As previously described , CFG is one of the main properties of programs that are utilized by vulnerabilities analyses. For this reason, CFG reconstruction is one of the first challenges in any binary analysis tool. For example, in the already mentioned research by C. Zhang et al.[26] , two common approaches are described, depending on the disassembler type used: while linear disassemblers are simpler and cover most of the instructions, though they are not very effective in case of indirect calls and jumps, recursive disassemblers are more complex but handle indirect calls and jumps more effectively.

`REV.NG` has the ability to recover the CFG and the function boundaries for binaries that do not have debug or any other additional information. It can also recover program data-flow using some custom analyses developed by `REV.NG` researchers. The output produced by `REV.NG` is an `LLVM` IR module that can be compiled or processed with standard `LLVM` passes, allowing to write architecture agnostic analyses. `REV.NG` workflow starts by searching executable portions in input binary, and then it goes on to reconstructing basic blocks of instructions. Each basic block can be labeled from different sources, i.e. program entry point, or harvested with custom analyses. When all basic blocks are labeled, their connections are reconstructed following direct pointer calls and jump tables in the global data sections (i.e., ".rodata" ELF section). After that, `REV.NG` starts to build the CFG of the binary: direct jumps can be translated easily into CFG nodes, while indirect branches and indirect function calls are more complex to analyze. It is not always practical to enumerate all possible indirect jumps for those cases, and `REV.NG` categorizes indirect branches and indirect function calls into three macro-cases:

- Compiler-Generated, function-local CFG
- Hand-written assembly
- Indirect function calls

At the moment, `REV.NG` aims at handling correctly only the first macro-case, while a lot of work still needs to be done in order to achieve proper manipulation of the other ones, even if they are often involved in many situations for function boundary detection. In fact, indirect function calls are typically implemented for virtual tables and used for example in C++ methods that can be overridden. Researchers tested `REV.NG` CFG reconstruction performances against most known decompilers and binary analysis tools and the results showed that `REV.NG` was able to recover more blocks than all other competitors but IDA pro. While IDA pro was able to reconstruct more code than `REV.NG` , researchers have noticed that `REV.NG` reconstructed CFG was in general more precise than IDA pro one.

2.5.2 Function-boundaries detection

Another important feature of `REV.NG` is *function-boundaries detection*: the tool identifies the entry points of function and assign to each of them the reached basic blocks. At the beginning, all discovered basic blocks are collected into a "pool" function called *root*, then after a subset of basic blocks is identified as the body of a specific function, the root is copied into a new function named either with the real name of the function, if debug symbols are present, or with the address of the starting basic block. This solution allows `REV.NG` to reconstruct also functions that can contain shared code inside their body. Thanks to the Offset Shifted Register Analysis and SET Analysis, `REV.NG` is able to reconstruct also the structure of function stack, and in many cases it is also able to understand which are the function parameters, independently from the call convention of the architecture

`REV.NG` was born as a thesis project developed inside our university by Alessandro Di Federico and Giovanni Agosta, and currently it is maintained by a large team of developers. Nowadays `REV.NG` has a stable code and also an ongoing development of a decompiler that can translate a executable binary into High Level C source code.

Recently the team behind `REV.NG` has released a new version that introduced supports for `LLVM` 9.

2.6 Project Goal

Since programs can be affected by multiple and different types of vulnerabilities, we decided to set buffer overflow vulnerability, which are among the most widespread ones, as target for our analyses.

The goal of this project is to develop a security tool capable of detecting functions that contain at least a buffer overflow vulnerability and that are reached by a user input inside binary programs. The final program should be able to identify such vulnerable functions in all type of binaries independently from the linking type (statically or dynamically linked libraries) and the architecture (i.e., x86 or ARM). Additionally, the tool should be able to understand when user input is propagate from one function to its callers.

In order to achieve this goal we will face many problems related to differences of available instructions sets architectures (or ISA). Even if machine code has some common features among all architectures (i.e., variable accessed by registers), there could be substantial differences between instructions of the same type belonging to two different ISAs. Our tool should be able to work transparently, no matter what instruction set is used in the binary. In our case, the main problems are:

- binaries programs make use of pointers to access memory variables. Pointers can be modified as any other variables of the program, so it can be difficult to track which memory regions are pointed by a pointer during execution. In our case, our tool should be able to understand which stores point to a memory area that represents a function argument or a function local variable to identify buffer overflows patterns.
- loops in binaries are represented by a sequence of branch and jump instructions. In many cases, jump instructions used in loops have a fixed target (typically a LABEL) but in some cases they can use indirect jumps too. Loop identification is much more complex when there are indirect jumps, but our tools should be able to correctly identify loops inside binary in order to detect buffer overflow vulnerabilities, in particular those that are controlled by user input.

`LLVM` IR hides many aspects to architecture-dependent machine code:

- `LLVM` IR offers also a standard way to represent loop structures that can be easily inspected by analyzers.
- all functions invocation can be represented in `LLVM` IR with the `call` instruction or the `invoke` instruction, without rules regarding which registers should be saved by caller.
- `LLVM` IR offers an infinite set of registers that could be used to access variables, reducing the possibility that more variables share the same register.

For this reason we decided to build our analysis on top of an intermediate representation of input binaries using `LLVM` IR.

In particular, we decided to use `REV.NG` inside our tool in order to reconstruct such intermediate representation of binaries compiled for different architectures. In fact `REV.NG` is able to understand calling conventions used in binaries, and convert them into the standard convention used in `LLVM` IR, recovering also the invocation of functions using the standard call instruction of `LLVM` IR. Additionally `REV.NG` is able to reconstruct binary functions along with their parameters and local variables.

Design and Implementation 3

As discussed in the previous chapter, our goal is to build a security tool that is able to identify buffer overflow vulnerabilities in binary executables. The tool should be able to handle binaries from many architectures and linked either statically or dynamically against external libraries. To achieve this goal we will use an approach based on the translation of the binary into an intermediate representation that is architectural independent, then we will develop analyses based on this intermediate representation that does not have to deal with architectural-dependent features. Thanks to this decision, our tool should be easily extended to support other architectures only by implementing the correct translation rules for that specific type of architecture.

The tool looks inside the intermediate representation of the binary for a particular pattern that represents buffer overflow and checks that the store destination inside this pattern depends on a function parameter or local variable. In particular, this pattern should contain a loop controlled by a variable value depending on a function parameter. Besides, this pattern is not coupled with any architectural-dependent characteristic.

After having identified the vulnerabilities, the tool will filter out functions that are not reached by user input, in order to exclude overflows that cannot be controlled by attackers. The tool implements a taint analysis on the intermediate representation using a list of input functions passed by tool user as source and then it should be able to propagate the input to callers. This process can be iteratively repeated adding marked callers to the initial list of input functions until a fixed point is reached. In particular, the tool must be aware of call loops inside the intermediate representation among different functions.

3.1 Design

We started the design process of our tool by defining a general structure that represents buffer overflow in common code that can be easily recognized inside `LLVM` IR.

We had to face different challenges during the progress of our project: while different analyses are able to perform advanced pointer aliasing, value-set inferring and memory accesses overlapping, they are not compatible with the `LLVM` IR produced by `REV.NG` lifting and are also quite challenging to re-implement inside `REV.NG` . The lack of such features forced us to design our project with a more general approach.

3.1.1 Buffer Overflow structure

For our analyses we defined that a buffer overflow is characterized by three properties that must hold in a program function:

- A buffer, or variable that can be controlled by user input
- A cycle whose iterations are controlled by user input
- Store Instructions inside that cycle that fill program memory with user controlled buffer values.

These property are translated easily into `LLVM` IR characteristics:

- A user controlled variable can be a function argument or a value coming from a particular system input function.
- A Loop inside the `LLVM` IR contains a specific component (called *Latch*) that represents the branch to the exit point of the loop or the next iteration. This component can be easily analyzed by `LLVM` passes, in particular the conditional instruction that rules the branch.
- `LLVM` has a specific type of Instruction for store, that is *StoreInst*, that offers different methods that help understanding where it is inside the module and what are its operands.

Once we had defined *what* can be a buffer overflow inside our tool, we started to work on *how* our tool could be able to actually recognize one when it is present. Since the desired result of our tool should be the correct identification of vulnerable functions (or portions of code) inside the input binary, we also defined when a function is vulnerable. For our analysis, a function can be considered vulnerable if it has these two properties:

- It must contain at least a *candidate loop*.
- Any of its candidate loops contain at least a risky store.

A *candidate loop* is a `LLVM` loop that is controlled by a condition instruction which uses at least one of the function parameters. A *risky store*, instead, is a store on variable that comes from a function argument.

In order to understand what comes from the user input, our tool implements a simple taint analysis that categorizes functions into two groups:

- *Input function*: system functions that read user input (like input streams or user's files) or wrapper functions that call these system functions and propagate the input with their parameters or results.
- *Non-input function*: all other functions that do not propagate user input

During the analyses, all functions that employ user input produced by discovered input functions are also called *marked functions.*

Analyses results will count only marked functions as vulnerable, because they could be effectively exploited by user input.

3.2 Implementation

The tool firstly recovers the `LLVM` IR from the input binary using `REV.NG` , and store it into a module named against the binary. Then it executes a pipeline of passes on this module. In figure 3.1 an high level overview of this pipeline is represented. The pipeline is composed by six passes executed in the following order:

- Max Steps Pass
- Backward Propagation Pass:
- Revng Function Params Pass
- Function Params Usage Pass
- Loop Dependencies Pass
- Security Wrapper Pass

Each pass inside the pipeline implements a specific step of the global analysis and its sub-result will be used by successive passes in the pipeline.

This approach, used also by `LLVM` in all its optimizations and analyses, allows our tool to schedule correctly the passes execution and also to parallelize the processing of each unit inside the `LLVM` module as much as possible.

In addition to that, with this pipeline the tool cache results from previous analyses with the same granularity, reducing the overall response time of the tool.

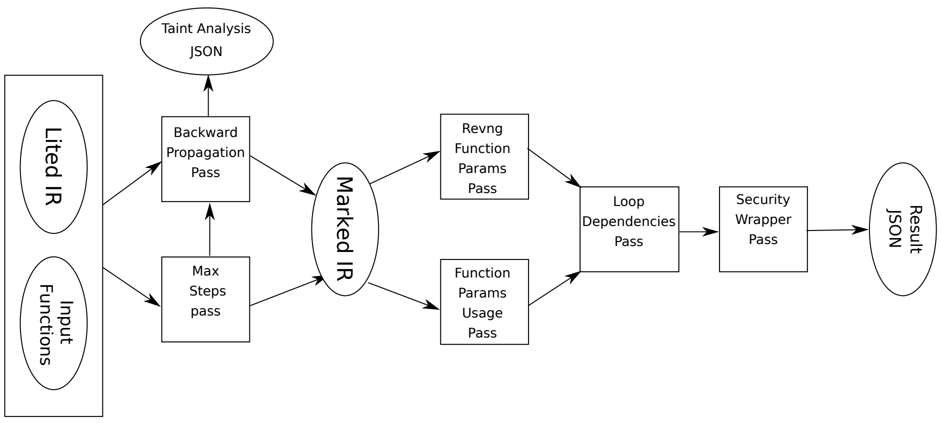

Figure 3.1: High-level design of the pass pipeline of the analysis

All tool passes are implemented as C++ libraries that are loaded dynamically by the `LLVM` optimization tool *opt*. This decision allowed also to use the powerful library provided by `LLVM` for command line options, that helped us extending passes with some configuration variables that can be change at tool invocation in order to tune the analysis output.

3.2.1 Preparation

During the first step of reconstruction of `LLVM` IR from the binary (also called lifting phase), our tool applies some optimization to the output module using `LLVM` passes and custom `REV.NG` passes. These passes simplifies the lifted IR for next analyses. Firstly, the output of the `REV.NG` lifting process does not isolate binary functions: functions can be isolated in the lifted `LLVM` IR applying the *Function Isolation* `REV.NG` pass. Additionally, `REV.NG` is able to detect ABI of the binary and enforce them to `LLVM` function, promoting CSVs used for function-passing as function arguments. This is achieved by executing the *Detect ABI* and *Enforce ABI* `REV.NG` passes.

Once a well structured `LLVM` IR is obtained, some additional `LLVM` passes are applied. The obtained result is restructured by applying *Loop Simplification* pass, already discussed in the previous chapter. In this way it is simpler for our analysis to understand Loop conditions and identify candidate loops.

3.2.2 Max Steps Pass

The first pass in the pipeline of our analysis tool is the *MaxStepsPass*. It is a simple CallGraphSCCPass that scans CallGraph of the input module for cycles. On each SCC inside the CallGraph it counts the number of function called. Then it counts the length of all branches from root node to all leaves node using a depth-first visiting algorithm. The result of this analysis is the maximum number found over all execution time. This simple analysis will be useful for the next transformation pass as iteration stop check.

3.2.3 Backward Propagation Pass

Next Pass in the pipeline is the Backward Propagation Transformation pass. It is a Module that propagates input values from a specific set of input functions to their caller, one layer at a time. The maximum number found by Max Steps Pass (Section 3.2.2) is used as upper bound of times that propagation will be repeated. This condition is used to avoid the pass to propagate the input values inside call cycles infinitely. The `LLVM` IR of the analysis is scanned for input functions or functions with name equal to address of relocation stub. Then, callers of these input functions are marked for following analyses. Each caller is then analyzed to check if they propagate their input with one of their arguments or with the return value. The process is reiterated to cover all CallGraph.

Handling relocations

While `REV.NG` out of the box is able to understand debug symbols and to reconstruct names of functions from binaries built using supported default toolchains, for binaries built outside those toolchains isolated functions are sometimes named with the address of entry basic block. This happens commonly with functions imported by dynamically linked library. For our Backward Propagation passes this incongruity means that it is not possible to identify input functions by their name so we had to design another approach to find them also by entry basic block address.

Fortunately, in the case of dynamically linked executable, the most common linkers— the system tool that is responsible to load required libraries and replace functions addresses inside call instructions — requires some information to process run-time dynamic linking and that information is typically embedded inside binary itself. For example, Executable and Linkable Format (also known as ELF) , used by GNU/Linux and other UNIX-like Operating Systems as executable binary format and shared libraries, contains specific sections dedicated to dynamic linking, i.e., ".plt" or ".got.plt" sections. Furthermore, GNU/Linux offers different tools to inspect ELF objects and also to parse out only specific sections.

In order to make our analyses cover also those type of binaries not fully supported by `REV.NG` , we built a component that is invoked before all analyses and that prints inside a file the entry point address of each function. Backward Propagation Pass then reads this file and loads all relocation addresses, and for each input function tries to find it inside the module by name or by address.

3.2.4 Revng Function Params

The *Revng Function Params* pass analyzes each function inside the module and collects some information about parameters and stack variables for each of them. It firstly scans metadata attached to each function by Enforce ABI and Function Isolation custom revng passes. These passes tries to understand function calls instruction and what are return types of functions and what are the functions arguments.

In `LLVM` , the standard way to define function stack-allocated variables is by using *Alloca* Instructions, but `REV.NG` is not always able to promote discovered variables to *Alloca* instructions. In addition to that, `REV.NG` custom passes in some cases are not able to recover function arguments from lifted instructions. In these cases, those variables are translated as normal uses of specific CSVs.

Revng Function Params pass tries to overcome these problems with some expensive solutions. First it tries to find those un-promoted function arguments searching directly for known CSVs used as argument-passing register in particular architectures. After that, Revng Function Params pass scrapes stack pointer register CSV uses and collect all instructions that convert a particular offset from register value into a pointer variable (in `LLVM` IR this process is represented by an *Inttoptr* instruction). The pass then store all the variables allocated on the stack by the function. Unfortunately these solutions tight analyses to subset of known register names used by common architectures and makes incomplete results on other architectures.

It also recognizes internal QEMU functions, which are not relevant for our analyses purpose, and build a structure with the information about each analyzed function that can be obtained by later passes.

3.2.5 Function Parameters Usage

Function Params Usage pass is used to build all value flows of function arguments and variables found by . The flows are collected and cached using a depth-first visiting of variable uses. Each first Use of a variable or parameter defines a new flow that is attached to the starting variable or parameter, creating a new vector populated initially by the related first User. During flows visiting, the pass keeps track of already visited Users, in order to avoid not-terminating cycles. The pass populate an initially empty queue with new users discovered for each use, adding only those users that were not already visited. When the queue is empty, the flow is finished and pass can start to analyze next flow or next variable or argument. This procedure is facilitated by SSA form of the `LLVM` IR, that makes def-use chains syntactically explicit.

The `LLVM` IR and search for their parameters that maybe revng was not able to isolate. It also scans the stack for parameters allocated on it. Then builds the def-use chains in SSA form for each of them. Then search in each chain for store that takes as argument a pointer to a function argument.

The Figure 3.2 shows an example of the flows collected on the variable "rsp" in a sample lifted function.

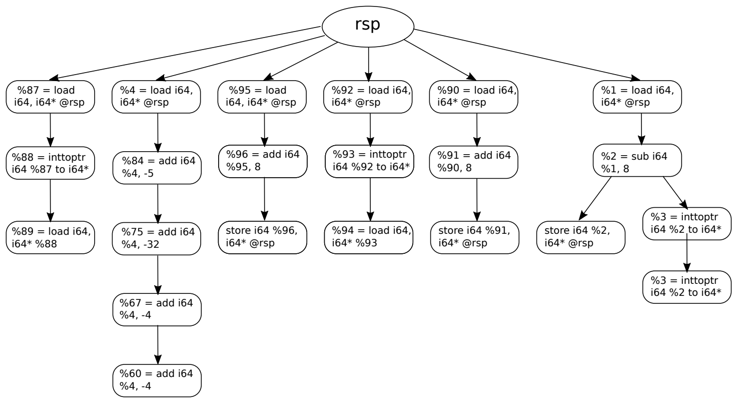

Figure 3.2: An example of a variable-flows produced by Function Params Usage Pass

At the end, the pass collects all risky stores idenfied for each flow, using the definition in Section 3.1, into a vector that can be later analyzed by other passes. This solution allows to understand when store instructions use pointers that derive from function arguments or local variables, solving the problem represented by pointers described at the end of the previous chapters.

With this approach, analysis results, specially risky stores, are cached for each function and later passes do not have to build and traverse again the def-use chains.

3.2.6 Loop Dependencies Pass

Loop Dependencies pass scans all the loops inside each function for loop conditions that can depend from a function argument. Thanks to Loop Simplification `LLVM` pass, we can approach all loops in a general way, and covering in these most of the possible loops types. Additionally, Loop Info Analysis pass builds for each function a Loop Info object for each loop: Loop Info is a wrapper class that contains all relevant information about the loop and also allows to easily access relevant component of the structure.

For each Loop Info, Loop Dependencies pass access the *Latch* block of the loop and scans Instructions inside it until it found a conditional statement, i.e., `icmp`. Then the pass analyzes statement operands and checks if they are contained inside any of the value flows built by . When this happens, the pass marks the related loop as *candidate loop* and search for risk stores that are inside it. If any of the candidate loops inside a function contain at least one risk store, the pass marks the function as vulnerable.

This pass, along with the ability of `REV.NG` of translating loops in binaries code, solves the loop identification problem defined at the end of the previous chapter.

3.2.7 Security Wrapper Pass

Last pass of the pipeline, Security Wrapper pass collects results from all other passes and re-factorizes them into a JSON object, with one sub-object for each function. This pass is used as synchronization point for the pipeline, because it have to wait that all previous passes are completed before writing result file. It also allows to user to choose which fields must be contained inside the JSON result: for example, a user can be not interested in many details about the analysis and can choose to include only result from pass 3.2.6 inside the result. This example solution can also speed up a lot the tool execution time because the I/O operations are the bottleneck of the program, in particular the dump of the JSON result into a file.

3.2.8 Analysis Output

The analysis produces two JSON output files: one contains taint analysis results and the other contains information about buffer overflows discovered.

Taint analysis JSON file contains the lists of marked functions inside the program as a key-value map: the key is the marked function and the value is a list of the input functions called by the marked function. For each input function, there is also additional fields that identify what is the parameter used to transfer the input.

The other JSON file contains the lists of marked functions analyzed by the program. For each of them, the file reports also the arguments recognized by RevngFunctionParamsPass, the chains recovered by FunctionParamsPass, the vulnerable loops and their conditions discovered by LoopDependencies-Pass, and the risky stores contained.

The use of the standard JSON format for the output of the analysis, allows also additional post-analyses refinement, like statistic collecting and filtering thanks to the versatility of such format.

Experiments 4

In this chapter we present the experiments we designed to test our tool abilities in buffer overflow vulnerability detection. The goals of these experiments are:

- to understand if the tool is actually able to identify real world buffer overflow vulnerabilities
- to understand if it can be used to discover new buffer overflow vulnerabilities
- to measure performances of vulnerability detection among different types of binaries

Test Environment For our tests, the tool was compiled following the standard build instructions of `REV.NG` . It has been installed on a GNU/Linux virtual machine equipped with a dual-core x86-64 virtual processor and 8 GB of virtual memory.

4.1 Sanity Check

The first experiment was designed to test the validity of our analyses in an ideal case and to identify a buffer overflow for a simple program containing a single vulnerable function. It was designed also to test tool robustness against binaries without buffer overflow vulnerabilties: the tool, of course, should not find buffer overflow in secure program. In this case we are not interested in functions reached by input, because the simple program does not require any user input. In order to test the correctness of our analysis we used three C simple programs written by us:

1. the first program, named "strcpy", was a simple reimplementation of a vulnerable strcpy
2. the second program, named "not-vulnerable", was a simple program without vulnerabilities

3. the third program, named “standard-loop”, was a simple program with a loop and no vulnerabilities.

The results of the validation tests are reported in Table 4.1

Binary	Vulnerable Functions	total functions	ground truth	Risky Stores	total stores
not-vulnerable	0	1	0	0	4
strcpy	1	10	1	4	55
standard-loop	0	7	0	0	41

Table 4.1: Results of the validation tests

From the results we notice that the analysis is able to identify our vulnerable function inside the program “strcpy”. On the other hand, for not-vulnerable programs the expected results were obtained: in “not-vulnerable” and “standard-loop” programs no risky stores were reported by analyses output, and even in “not-vulnerable” programs no loop were detected. Therefore the experiment demonstrated that our tool is able to recognize buffer overflow pattern in our designated target, which is the vulnerable function in “strcpy” program.

4.2 DARPA Challenges binaries

In the second experiment we decided to test our tool abilities with uncommon binaries, using the binaries shipped with the DARPA Cyber Grand Challenge (or CGC) platform. The DARPA Cyber Grand Challenge is a competition designed to help and test automatic defenses systems in computer technologies. DARPA hosted the Cyber Grand Challenge Final Event on August 2016 in Las Vegas and then released the developed technologies publicly, enabling researchers all over the world to benefitfrom them. In particular these technologies consist of a custom platform containing a set of vulnerable binaries that could be exploited by attackers. This platform now is still used to test new computer security tools. This test suite has also become one of the standard benchmarks used when new computer security tools are released. In our experiment we collected different binaries provided by the CGC platform, either vulnerable to buffer overflow or not, and packed each of them into an ELF format in order to make them compatible with `REV.NG` . Then each binary was analyzed following the same approach of the experiment described in Section 4.1, with the addition that we also provided a list of input functions (in particular common C functions used to access user input) to our tool in order to trigger the taint analysis filtering.

Table 4.2 reports the results of the tests conducted on the CGC binaries marked with at least one buffer overflow vulnerability.

Binary	Vulnerable Functions	Marked Functions	Overall Functions	Risky Store	Total Stores	Contains BO?
CADET_00001	3	0	8	38	218	yes
CADET_00002	7	0	16	64	555	yes
CROMU_00007	16	0	45	147	1221	yes
CROMU_00013	18	0	58	546	2218	yes
KPRCA_00001	7	0	41	7	908	yes
KPRCA_00003	9	0	24	100	654	yes
NRFIN_00003	10	0	36	131	1180	yes
YAN01_00001	6	0	18	288	1025	yes
YAN01_00004	16	0	51	360	1700	yes
YAN01_00005	10	0	40	315	1321	yes
CROMU_00002	16	0	41	432	1219	no
CROMU_00027	27	0	66	744	2378	no
CROMU_00029	24	0	69	633	2611	no
CROMU_00041	29	0	49	260	1450	no
KPRCA_00013	48	0	112	1134	4691	no
KPRCA_00023	15	0	79	130	1472	no

Table 4.2: Experiment results on the binaries extracted from DARPA CGC binaries

The first observation we can draw is that our taint analysis does not work with this type of binaries, hence we analyzed also results for non-marked vulnerable functions. Secondly, our tool found buffer overflow patterns in different functions for binaries that were declared in the CGC platform as vulnerable against buffer overflows. The downside is that the tool found similar patterns also in many functions of binaries that were not known to contain a buffer overflow (even if this cannot completely exclude that they may contain a buffer overflow vulnerability). In addition, we noticed that for this type of binaries the approach used for taint analyses is not useful because they binaries are compiled on top of an operating system which contains different system functions for input operations. But since these binaries do not contain debug symbols or useful symbol tables, we are not able to identify manually those functions.

These results show that even in binaries without buffer overflow there is a pattern similar to buffer overflow and, unfortunately, they demonstrate that our results may contain a larger number of false positives when taint analysis filtering is not used.

4.3 Real World Binaries

This experemինent was designed to test performances of our tool in a realistic scenario, with much more complex programs. As target program we used a list of binaries reported as vulnerable in different public vulnerabilities lists, more specifically, we referred to The Ultimate security vulnerability datasource, Exploit Database and Common Vulnerabilities and Exposures (CVE) List as sources of vulnerable programs. Each program was analyzed in the same way as binarieso of the experiment described in Section 4.2, but we also added some common functions used in the C++ language to the list of input functions . For each binary, we check if the tool found had identified the vulnerable function listed in the corresponding CVE. Whenever the CVE did not report the vulnerable functions, we made some assumptions: if a function among the vulnerable ones had a name that could be reconduced to the CVE description, then the CVE was verified; otherwise if none of the vulnerable functions had a name attributable to CVE description, the CVE was not verified.

The results of the experiment on real world binaries are reported in the Table 4.3.

CVE	Binary	Vuln. Functions	Marked Vuln. Func.	Ground Truth	Analyzed Functions	Risky Stores	Total Stores	Verified
CVE-2014-0158	OpenJPEG	115	0	3	348	3549	12306	yes
CVE-2008-1959	yespp	109	0	1	2261	1675	41169	yes
EDB-ID-42357	mawk	35	0	?	190	492	6552	yes
CVE-2019-1010057	nfdump	28	0	2	174	628	6917	yes
CVE-2004-1257	abc2mtex	27	0	1	119	851	3255	yes
CVE-2019-14267	pdfresurrect	18	0	1	83	434	1484	yes
CVE-2000-0359	thttpd	14	0	?	200	79	3452	yes
CVE-2018-20337	LibRAW	11	0	2	375	261	1083	yes
CVE-2019-1010301	jhead	8	0	1	187	87	6318	yes
EDB-ID-46807	MiniFTP	8	0	?	157	58	1726	yes
CVE-2018-17174	nmealib	5	0	1	49	82	1103	no
CVE-2017-6438	libplist	4	0	3	96	46	1335	no
CVE-2018-1000221	pkgconf	3	0	1	87	84	5421	yes
CVE-2019-3574	img2sixel	0	0	2	23	0	43	no
CVE-2019-16346	ngiflib	0	0	?	32	0	797	no

Table 4.3: Experiments results on the real world vulnerable programs

Similarly to the experiment described in Section 4.2, also in these tests our taint analysis did not work very well with dynamically linked binaries for x86 and x86-64 architectures, thus we took into account also results for non-marked vulnerable functions. This experiment demonstrated that our tool is able to detect and verify 10 CVEs among the test binaries, identifing the vulnerable functions reported in the description of each CVE. The experiment shows also that our tool needs improvements in filtering out false positives, because the number of vulnerbale functions detected was generally bigger than the one reported in CVEs. Another important observation provided by the experiment is that our tool performs better on smaller input binaries: the results report a smaller percentage of detected functions in binaries with a number of scanned functions in the order of 100/200 functions. Unfortunately though, the results also show that our tool is not able to identify buffer overflow in all possible inputs, hence some CVEs could not be verified with our experiment. In order to have a better understanding of this last problem, we manually checked the lifted IR produced by the tool for each binary and we found out that `REV.NG` was not able to reconstruct calls to low level functions with standard `LLVM` call instructions. In particular, we noticed that taint analysis marked a common function among all binaries (that did not contain buffer overflows pattern and so it was not listed in the results), named “do_syscall”. This name suggests that `REV.NG` may handle call to these low level functions with this auxiliary function.

Lastly, it is interesting to underline the case represented by the binaries “img2sixel” and “ngiflib” where the tool did not find any buffer overflow pattern. In fact, in those cases the results show that the tool is capable of correctly recognizing functions inside the binary, but the number of risky stores suggests that there might have been a problem on the analyses of loops control instructions inside those binaries.

4.4 Unknown binaries

The follwing experiment is designed with the purpose of testing our tool abilities to discover new vulnerabilities in binary deployed without source code, therefore we collected binaries extracted ARM-based router firmwares, using the tool developed by P. De Nicolao et. al in their project ELISA [7]. While most of theses firmware are based on Linux operating system, they may use different formats for binary executables. Fortunately, the ELF format can be used also as an archive for binary data, therefore we were able to easily pack each binary found in the firmwares into an ELF executable, making it compatible with our tool. Unfortunately, since these binaries had not yet been analyzed by anyone, we did not have a ground truth to use as performance standard for our tool, therefore we manually analyzed only binaries in which the results reported at least one vulnerable function reached by input functions. We used IDA Pro tool to decompile each binary and then look at the decompiled code to double-check each vulnerable function found by the tool. We chose IDA Pro because it represents the de-facto state of the art decompiler in the current market, and it is commonely used to identify vulnerabilities in binaries, particularly loop based buffer overflow as reported by S. Rawat and L. Mounier in their work [18] on identification of buffer overflows in binaries.

The results of the experiment are reported in the Table 4.4.

The results show that our tool was able to identify vulnerable functions in almost every input binary. Additionally, all the binaries were marked as reached by input functions from the anlaysis, revealing a possibility to exploit such vulnerabilities by a malicious attackers. Our manual check performed on

Binary	Vulnerable Functions	Marked Functions	Analyzed Functions	Risky Stores	Tot Stores	Confirmed
mDNSResponderPosix	176	104	786	2044	7392	no
afpd	55	44	1069	358	3037	no
app_data_center	26	10	309	385	1595	no
vol_id	5	2	77	9	16	no
UsbIppCheck	2	2	29	12	25	yes
setfattr	4	1	49	16	36	no
pidstat	12	9	159	513	998	no

Table 4.4: Experiment results on the binaries extracted from router firmwares

these binaries from discarded the results regarding all binaries but one, that is “UsbIppCheck”, which was the only one to contain a real buffer overflow vulnerability. Thanks to this experiment, we proved that our tool is useful for discovering vulnerabilities inside binaries without source code.Moreover, it also demonstrated that our tool is able to recognize vulnerabilities in different types of binaries for different architectures.

4.5 Taint Analysis Performances

This last experiment was designed to mesaure the performance of our tool in filtering out functions that were not reached by user input. In this test we collected the results regarding taint analyses from all the previous experiments, as well as some additional binaries extracted from firmwares which were not manually checked. Data about filtered vulnerable functions for each binary are reported in Tables 4.5, 4.6, and 4.7.

Binary	Vulnerable Functions	Marked Vulnerable Functions	Analyzed Functions
OpenJPEG	115	0	348
sipp	109	0	2261
mawk	35	0	190
nfdump	28	0	174
abc2mtex	27	0	119
pdfresurrect	18	0	83
thttpd	14	0	200
LibRAW	11	0	375
jhead	8	0	187
MiniFTP	8	0	157
nmealib	5	0	49
libplist	4	0	96
pkgconf	3	0	87
img2sixel	0	0	23
ngiflib	0	0	32

Table 4.5: This table reports the performances of taint analysis on binaries of the experiment described in Section 4.3

Binary	Vulnerable Functions	Marked Vulnerable Functions	Analyzed Functions
mDNSResponderPosix	176	104	786
afpd	55	44	1069
wl	50	18	1004
tc	49	39	789
business_proc	35	7	777
6relayd	31	25	208
app_data_center	26	10	309
px5g	26	12	252
wtfslhd	24	14	286
athdiag	21	2	286
ozker	21	8	351
partx	19	2	339
wimaxd	13	7	443
pidstat	12	9	159
vol_id	5	2	77
setfattr	4	1	49
qosd	4	2	165
UsbIppCheck	2	2	29

Table 4.6: This table reports the performances of taint analysis on binaries of the experiment described in Section 4.4 and other extracted from the same firmware

Binary	Vulnerable Functions	Marked Vulnerable Functions	Analyzed Functions
CADET_00001	3	0	8
CADET_00002	7	0	16
CROMU_00007	16	0	45
CROMU_00013	18	0	58
KPRCA_00001	7	0	41
KPRCA_00003	9	0	24
NRFIN_00003	10	0	36
YAN01_00001	6	0	18
YAN01_00004	16	0	51
YAN01_00005	10	0	40
CROMU_00002	16	0	41
CROMU_00027	27	0	66
CROMU_00029	24	0	69
CROMU_00041	29	0	49
KPRCA_00013	48	0	112
KPRCA_00023	15	0	79

Table 4.7: This table reports the performances of taint analysis on binaries of the experiment described Section 4.2

The combined results from all four experiments demonstrate that our taint analysis is effective only on binaries which were taken from router firmwares used in the experiment described in Section 4.4. The reason behind such increase in performances could be due to the direct use of low level functions in firmware binaries, without using any abstraction provided neither by an operating system nor by a high-level programming language. Besides, the drop in performances for CVE binaries could be explained in a way similar to the one discussed at the end of Section 4.2, where operating systems in collaboration with high-level languages and compilers provide abstractions that are not effectively handled by `REV.NG` function isolation feature, forcing it to use auxiliary functionalities that disrupt the standard call interface of `LLVM` IR.

The results for DARPA CGC binaries instead are not covered by taint analysis functionalities, because they use a different set of system functions: we manually inspected many CGC binaries to understand which functions could be marked as starting input ones, but because of the lack of symbol tables and debug symbols (probably due to the fact that they were not originally packed in ELF format) we were not able to identify any input function.

Related Works 5

In this chapter we present some related works about static binary analysis that can be used as reference for future improvements on this project.

5.1 Static Value Flow Analysis

Static Value Flow is a project conducted by Sui et al. [23], who produced a tool for Static Value Flows analyses based on `LLVM` . It was designed to construct data-flow analyses from `LLVM` IR of C and C++ programs, building an abstraction layer that can be extended easily by any other developer. The tool offers a set of *points-to* analyses, a particular type of analysis used to reconstruct the data-flow of variables taking into account also pointers. Each points-to analysis is composed by a *Graph*, a set of *Rules* and a *Solver*. Each of these components has a specific role and is loosely coupled to the others, allowing to easily build new customized analyses: the *Graph* builds an high-level abstraction of the `LLVM` , marking portions where pointer analyses will be executed; the set of *Rules* defines how information can be obtained by each instruction; and the *Solver* identifies the constraints and defines in which order they must be solved.

The tool offers the Andersen's analysis as default pointer analysis: it was built following the instructions reported in Andersen's study about analysis of C programming language [1] and using a *Solver* based on Wave analysis [17], while as additional analyses some field-sensitive flow analyses, able to track also flow of values inside object fields for C++ programs.

The results of the Pointer analyses are reused by the tool for the *Mod-Ref Analysis*, that catches inter-procedural references and uses of variables. This analysis partitions the memory in different regions, using also the MemorySSA transformation (that will be discussed later), reconstructing also indirect references and uses. After that a *Value-Flow graph* (VFG) is built, connecting each variable definition to all its uses known so far. The VFG can be used from other analyses to access def-use chains of all variables in the program.

The versatility of the SVF tool allows its uses in different scenarios: in addition to simple pointer analyses, it can be useful in taint analysis based on source-sink path discovery or also for accelerating dynamic analysis, removing unnecessary instrumentation and reducing run-time overhead.

5.2 Monotone Framework

There are many researches about data flow analyses based on mathematical tools, but one of the most powerful is the *monotone data-flow analysis framework* developed by Kam, John B. and Ullman, Jeffrey D. [9]. It is a generalization of the Kildall's lattice theoretic, adapted for DFG. The framework is built around the mathematical concept of *flow graph*, *semilattice* and *meet* operation. While we have already explained what flow-graph is, we still need to define what a *semilattice* and a meet *operation* on it are. Kam, John B. and Ullman, Jeffrey D, in their paper, define a semilattice with a meet operation, as following:

Definition. *A* semilattice *is a set L with a binary* meet *operation $\triangle$ such that for all $a, b, c \in L$:*

$$\begin{aligned} a \triangle a &= a && (\textit{idempotent}) \\ a \triangle b &= bAa && (\textit{commutative}) \\ a \triangle (b \triangle c) &= (a \triangle b) \triangle c && (\textit{associative}) \end{aligned}$$

In such a semilattice, thanks to the meet operation it is possible to define an order relation $>$ among the elements of L: we say that given $a, b \in L$

$$\begin{aligned} a \geq b &\qquad \text{iff} \quad a \triangle b = b \\ a > b &\qquad \text{iff} \quad a \triangle b = b \wedge a \neq b \end{aligned}$$

The order relation $>$ allows then to define a lower and an upper bound for the semilattice.In particular, in each semilattice L we can define two particular elements: the *zero element* 0, such that $\forall x \in L \quad 0 \triangle x = 0$, and the *one element* 1, such that $\forall x \in L \quad 1 \triangle x = x$

For each semilattice L it is possible to construct the *monotone function space associated with* L, that is a set of particular functions on L. In the same paper the following definition of a monotone function space associated with a semilattice is reported:

Definition. *Given a bounded semilattice L, a set of functions F on L is said to be a* monotone function space associated with L *if the following conditions are satisfied:*

1. *Each function $f \in F$ satisfies the monotonicity condition,*

$$\forall x, y \in L, f \in F \quad f(x \triangle y) \leq f(x) \triangle f(y)$$

2. *There exists an identity function i in F, such that*

$$\forall x \in L \quad i(x) = x$$

3. *F is closed under composition, i.e., $f, g \in F \Rightarrow fg \in F$, where*

$$\forall x, y \in L \quad fg(x) = f(g(x))$$

4. *L is equal to the closure of $\{0\}$ under the meet operation and application of functions in F.*

Given the definition of these needed concepts, the paper then formalizes the concept of a *monotone data flow analysis framework* with the following definition:

Definition. *A Monotone data flow analysis framework is a triple $D = (L, \triangle, F)$, where*

1. *L is a bounded semilattice with meet $\triangle$*

2. *F is a monotone function space associated with L*

A Monotone framework can be used to state the most common problems that are targeted by data-flow analysis, from simpler ones like variables oddity at given instruction to much more complex ones like the Abstract Interpretation for program proposed by Cousot et al. [4]

In order to use a monotone data flow analyses framework, users only have to state the problem they have to face using a semilattice L and a monotone space function associated with it according to its goals. While this task may not seem particularly difficult, actually the designing of a semilattice is a very demanding job, because of the complex and various properties the set of values must satisfy. Alongside to the growth in the number of values, also the process to verify that the properties of the monotone space associated with the lattice are satisfied becomes increasingly difficult.

Once the problem has been stated, the framework can be solved using two possible algorithms. The most common algorithm used to solve Monotone Data Flow frameworks problems is the *meet over all path solution* or MOP algorithm: it tries to propagate the information from $\{0\}$ at the starting point to a given m node in the graph, meet-ing all the solutions found over

all possible paths that connect n_0 to m. The problem with this algorithm is that it is known to be undecidable, hence there is no certainty on its termination. The other well known algorithm used to solve monotone framework based problems is the *maximum fixed point solution* or MFP algorithm. This algorithm sorts the nodes that must be visited to reach the given node m in reverse-post order, and it always takes the first node to appear on the list.
The MFP algorithm proved to be decidable,therefore it will always terminate; this quality though has its own costs: in order to be used as solving algorithm MFP requires the meet operation in the semilattice to satisfy also the distributive property. This additional requirement increases the difficulty of the monotone data-flow framework design process.

6 Limitations

The experiments conducted on our tool exposes different problematics that the tool is not able to face. While some of them are structurally impractical to solve, others are related to the effort required to solve them and they are untargetable by a single-man thesis project.

6.1 Structural limitations

The first observation that arises when looking at the experiments results, is the drop in accuracy performance when the tool analyzes binaries compiled with other toolchains than the one shipped with `REV.NG` . This poor performances are caused mainly by the quality of the lifting result: while `REV.NG` is able to track different optimizations and transformations performed by compilers shipped with it, its efficacy gets much worse when it has to deal with other compilers. This affected mainly the ability to detect and enforce the ABI of the binary, decreasing the number of arguments that are promoted in isolated functions. Additionally, when `REV.NG` tackles such binaries, the symbols are not always well interpreted, and the consequence is that many functions are labeled incorrectly or, even worse, different fucntions bodies are grouped inside a single macro-function.

There is also another problem that is quite impractical to tackle in binary analysis: dynamically linked functions. It is impossible to understand if one of these functions may be vulnerable beacuse their code is not actually present in the binary, instead it is available only at run-time. In different cases it is possible to rebuild the environment in which the binary should be executed, like the many approaches used in dynamic binary analysis, but there are also many cases in which this is an impractical solution. In addition to that, linked functions are not always the same in all possible environments because of the behavior of dynamically linked libraries: this solution has been proposed to decouple the libraries from the user binaries, allowing to update them indipendentely. In this case it becomes useless to analyze all linked libraries, because some of them may be vulnerable depending on the systems in which the binary is executed.

Compiler optimizations applied to binary can also decrease the performance of our tools. In most cases these optimization changes in a disruptive way the structure of the original code in order to produce the binary. One of the most common optimization that has decreased the performance of our tests is the inlining optimization: compilers can replace the call to a function inside another function with the actual body of the called function, removing the cost of an additional function call. The biggest problem of this optimization process is that callee-saved and caller-saved registers are no more required, and the compiler can choose to use different registers for the called-function parameters. In these cases, Function Params Pass may not include the def-chain for the called functions inside the chains of the function argument, leaving many possible risky stores outside the result.

There are also cases in which our buffer overflow structure is not present for real vulnerabilities. Another common case is when a system or low-level function is used to perform a buffer copy, hiding the vulnerability inside the body of these functions. Most common case is represented by legacy binary that calls the already known vulnerable function “strcpy” shipped with GNU libc system library, whose buffer overflow exploit is described in a paper by Lhee et al. [13] about format string overflows.

6.2 Weeknesses

Another factor that affects our `REV.NG` version perfomances is that the Lazy Value Info analysis used to reconstruct variable value ranges is way too conservative. Among the tests conducted, Lazy Value Info analysis, for most of the store instructions marked as risky stores, returned the full set of value that an integer pointer can have (depending on the register size of the binary architecture) therefore making the result useless for us. For this reason, our tool treats as candidate risky stores also instructions that do not really represent a possible vulnerability. While there are many implementations of pointers-to analyses based on the `LLVM` IR, i.e., the already mentioned SVF, we had many problems in making them cover also IR produced by `REV.NG` lifting. These problems let us stuck with the simpler and less precise Lazy Value Info analysis of `LLVM` .

7 Conclusions and Future Works

The experiments exposed in the previous chapter have proven the ability of `REV.NG` to identify a common vulnerability like buffer overflow in binaries thanks to our modifications. While the results could become much better with additional work, it is remarkable how the `LLVM` framework simplified the implementation of such a difficult problem, mainly thanks to its ability to divide the work in isolated steps that can be automatically managed and orchestrated by the framework itself. Additionally, the ability of `REV.NG` to reconstruct a reliable CFG allowed able to target different architectures, with very few adjustements and only in particular cases like non-ELF binaries. The actual disadvantage of `REV.NG` is its tighten coupling with its shipped toolchains and the uncertainty given outside this ideal environment. The team is already working on enhancing and improving these weak points of the tool and the latest release of `REV.NG` already offers a better compatibility to all binaries compiled with a GNU C Compiler version lower than nine.

Other projects have also proved how powerful the `LLVM` framework is and what it is capable of, which makes us think that there might be a wide amount of enhancement that can be introduced on our version of `REV.NG` . Another actual disadvantage of `REV.NG` is that it requires a lot of space on the system to be compiled, installed and used because of its dependencies on QEMU engine and `LLVM` framework. While at this moment there is no pre-packed version of `REV.NG` that can be downloaded and executed directly without compiling it, it would be amazing to make it less strictly coupled with its dependencies, for example without the requirement of recompiling them any time that `REV.NG` is compiled. To do so, the `REV.NG` team has to make `REV.NG` capable of working better with standard implementations of QEMU and `LLVM` since, as of today, it requires to apply some patch to these two frameworks before they are compiled, breaking the compatibility with their mainstream versions.

Regarding the actual security analyses, it would be great to integrate them with much more complex analysis tools to understand better how values are propagated through variable: the best solution would be to design and implement a monotone data flow framework that is able to understand which ranges of values each variable can hold for each node in the CFG. Another approach would be to reimplement the passes developed by SVF and make them compatible with the output of the `REV.NG` lifting, in order to understand when different pointers may point to the same memory area or on overlapping memory-areas, leading to memory corruption or buffer overflows.

Despite the poor performances in particular cases, we are quite satisfied with the results achieved by our project and we hope that with additional work it can get more precise.

Bibliography

[1] L. O. Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. University of Cophenhagen, 1994.

[2] S. Bekrar et al. “A Taint Based Approach for Smart Fuzzing”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* 2012, pp. 818–825. DOI: `10.1109/ICST.2012.182`.

[3] D. Brumley et al. “BAP: A Binary Analysis Platform”. In: *Computer Aided Verification.* Ed. by G. Gopalakrishnan and S. Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469. ISBN: 978-3-642-22110-1.

[4] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL ’77. Los Angeles, California: Association for Computing Machinery, 1977, 238–252. ISBN: 9781450373500. DOI: `10.1145/512950.512973`. URL: `https://doi.org/10.1145/512950.512973`.

[5] C. Cowan et al. “Buffer overflows: attacks and defenses for the vulnerability of the decade”. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00.* Vol. 2. 2000, 119–129 vol.2. DOI: `10.1109/DISCEX.2000.821514`.

[6] C. Cowan et al. “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” In: *USENIX security symposium.* Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[7] P. De Nicolao et al. “ELISA: ELiciting ISA of Raw Binaries for Fine-Grained Code and Data Separation”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment.* Ed. by C. Giuffrida, S. Bardin, and G. Blanc. Cham: Springer International Publishing, 2018, pp. 351–371. ISBN: 978-3-319-93411-2.

[8] A. Di Federico, P. Fezzardi, and G. Agosta. “rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery”. In: *2018 International Carnahan Conference on Security Technology (ICCST).* 2018, pp. 1–5. DOI: `10.1109/CCST.2018.8585654`.

[9] J. B. Kam and J. D. Ullman. “Monotone data flow analysis frameworks”. In: *Acta informatica* 7.3 (1977), pp. 305–317.

[10] S. Kim et al. “Testing Intermediate Representations for Binary Analysis”. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering.* ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 353–364. ISBN: 978-1-5386-2684-9. URL: `http://dl.acm.org/citation.cfm?id=3155562.3155609`.

[11] J. Křoustek, P. Matula, and P Zemek. *Retdec: An open-source machine-code decompiler.* 2017.

[12] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: `10.1109/CGO.2004.1281665`.

[13] K.-S. Lhee and S. J. Chapin. “Buffer overflow and format string overflow vulnerabilities”. In: *Software: Practice and Experience* 33.5 (2003), pp. 423–460. DOI: `10.1002/spe.515`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.515`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.515`.

[14] N. Nethercote and A. Mycroft. “Redux: A dynamic dataflow tracer”. In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003), pp. 149–170.

[15] N. Nethercote and J. Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *SIGPLAN Not.* 42.6 (June 2007), 89–100. ISSN: 0362-1340. DOI: `10.1145/1273442.1250746`. URL: `https://doi.org/10.1145/1273442.1250746`.

[16] J. Newsome and D. X. Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.” In: *NDSS.* Vol. 5. Citeseer. 2005, pp. 3–4.

[17] F. M. Q. Pereira and D. Berlin. “Wave Propagation and Deep Propagation for Pointer Analysis”. In: *2009 International Symposium on Code Generation and Optimization.* 2009, pp. 126–135. DOI: `10.1109/CGO.2009.9`.

[18] S. Rawat and L. Mounier. “Finding Buffer Overflow Inducing Loops in Binary Executables”. In: *2012 IEEE Sixth International Conference on Software Security and Reliability.* 2012, pp. 177–186. DOI: `10.1109/SERE.2012.30`.

[19] K. Serebryany. “OSS-Fuzz-Google’s continuous fuzzing service for open source software”. In: (2017).

[20] J. Seward and N. Nethercote. “Using Valgrind to Detect Undefined Value Errors with Bit-Precision.” In: *USENIX Annual Technical Conference, General Track.* 2005, pp. 17–30.

[21] Y. Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *2016 IEEE Symposium on Security and Privacy (SP).* 2016, pp. 138–157. DOI: `10.1109/SP.2016.17`.

[22] D. Song et al. “BitBlaze: A New Approach to Computer Security via Binary Analysis”. In: *Information Systems Security.* Ed. by R. Sekar and A. K. Pujari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–25. ISBN: 978-3-540-89862-7.

[23] Y. Sui and J. Xue. “SVF: interprocedural static value-flow analysis in LLVM”. In: Mar. 2016, pp. 265–266. DOI: `10.1145/2892208.2892235`.

[24] R. Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[25] M. Zalewski. *American fuzzy lop.* 2014.

[26] C. Zhang et al. “Practical Control Flow Integrity and Randomization for Binary Executables”. In: *2013 IEEE Symposium on Security and Privacy.* 2013, pp. 559–573. DOI: `10.1109/SP.2013.44`.