Reverse-engineering of binaries in a single execution : a lightweight function-grained dynamic analysis

Franck De Goër de Herve

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Franck de Goër de Herve

Thèse dirigée par **Roland Groz**, Professeur, INP, et co-encadrée par Laurent Mounier, Maître de conférences, Université Grenoble Alpes

préparée au sein du **Laboratoire Informatique de Grenoble** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Rétro-ingénierie de programmes binaires en une exécution

Thèse soutenue publiquement le **20 octobre 2017**, devant le jury présidé par Monsieur **Kavé Salamatian** et composé de :

Monsieur **Roland Groz**
Professeur, Grenoble INP, Directeur de these
Monsieur **Laurent Mounier**
Maître de conférences, Université Grenoble Alpes, Co-encadrant de these
Monsieur **Kavé Salamatian**
Professeur des universités, Université Savoie, Président
Madame **Valérie Viet Triem Tong**
Professeur associé, Centrale Supelec, Rapporteur
Monsieur **Jacques Klein**
Senior Research scientist, Université du Luxembourg, Rapporteur
Monsieur **Andy King**
Professeur, University of Kent, Examinateur
Madame **Sarah Zennou**
Ingénieur de recherche, Airbus Group, Examinateur
Madame **Marion Videau**
Maître de conférences, LORIA, Examinateur

2

Acknowledgment
Thank you. For the crosswords, the share of movies and books, for the ski and climbing, the millefeuilles, the help. For the card games, the hospitality, the bad jokes, the long discussions. The trips, the drinks. Thank you for your music. Thank you for your patience. For your time, for your advices, for the endless questions.
Thank you for the yaourt.

Extended abstract

In computer science, it may be important to analyze a program behavior, for integration, for testing, and nowadays often for security purposes (detection of vulnerabilities in legitimate programs, detection of malicious programs, etc.). Analyzing programs is (often) easier to do from source code than at binary level, as more information is directly available to the analyst, but sometimes the only thing we have is a binary file. This can be the case with commercial products as well as with malwares for instance. In addition, binary files include another kind of information not embedded at source-level (*e.g.*, concrete addresses, compilator choices, etc.). Finally, in some contexts, one favors the binary level as it embeds the code that is actually executed by the machine. Because of the *What You See Is Not What You eXecute* phenomenon, it can be different from what was expressed at source code (in particular if the compiler performs optimizations). For these three reasons, methods have been developed to work from a binary file without relying on source code.

This work introduces a new dynamic approach to analyze binary programs. We propose analyses in a reverse-engineering context with motivations related to security: finding vulnerabilities in programs, understanding their memory management and use, etc. We focus on retrieving high-level information of two kinds: structural information and behavioral information at the grain of functions. For structural information, we target function prototypes including arity and type of parameters. For behavioral information, we analyze data-flows of parameters between functions, and especially relatively to addresses. The main novelty of our approach is to propose analyses in a single execution, lightly instrumented, and based on heuristics. Our approach is designed according to three main criteria. First, it relies on weak assumptions: it neither relies on the source code nor on the symbol table of the program, thus it can be applied to closed-source stripped binaries. Furthermore, it

is not specific to a given type of machine code. We call this criterion *universality*. Second, we propose a lightweight approach to be *scalable*: our approach can be applied to common programs such as PDF viewers and text editors with a limited overhead. Third, we aim to propose an *accurate* approach, and, when possible, favor soundness over completeness: heuristics we make can lead to errors, but they aim to avoid false positives.

Our first contribution is to propose an approach in order to retrieve structural information from binaries, and in particular function prototypes as they may exist at source-level. Because we do not assume that the binary under analysis was produced by compilation, it requires to define the notions of function, call and return, input and output parameter at assembly level. We propose such definitions in this work that are, in addition, consistent with the corresponding notion at source-level in the case of compiled programs, but that include other binaries (for instance, binaries from hand-written assembly code and even binaries with no static representation). The second contribution of our work is to provide methods to retrieve behavioral information related to memory. We introduce a new notion we name *coupling*, which emphasizes the interactions between functions. More precisely, coupling describes address data-flows between two functions. From one execution, we retrieve couples of functions according to a coupling rate. We also address the problem of retrieving allocators from a binary, especially when it embeds a custom allocator instead of using a standard one (*e.g.*, `malloc` and `free` from `libc`). We propose a general definition of resource allocators (*e.g.* time allocators, multi-process schedulers, etc.), and a heuristic-based approach to retrieve memory allocators in binaries. Finally, we propose an open-source implementation of our approach, named `scat`, available on GitHub (see `https://github.com/Frky/scat`). `Scat` implements each analysis mentioned in this work, plus additional features for test purposes: each of the experimental results we present in this work can be replayed using `scat` (our benchmarks are provided, as well as the commands used to obtain the results and the charts).

Our experimental results show that our approach is accurate: in average, the arity detection presents a success rate of 93%, the type detection (although we retrieve a restricted subset of types) has a success rate of 96% on parameters and 91% on return values, and we do retrieve allocators in programs using the standard `libc` allocator as well as in several programs embedding their own allocators. Last, it is scalable, as we can perform analyses on common programs within a few seconds in most cases.

The thesis is divided in three parts. In a first part, we present the context of reverse-engineering we work in. In Chapter 1, we introduce the reverse-engineering in general. In Chapter 2, we focus on reversing binaries: its specificities, the kind of information to be retrieved and some existing works in this field. Chapter 3 presents two major approaches to analyze binaries: static and dynamic analysis. Chapter 4 states the problem we address. In a second part, we present our approach. In Chapter 5, we define the elements we target (functions, parameters, coupling, allocators). Chapter 6 presents our analysis relative to structural inference (*i.e.*, to recover arity and types of functions). In Chapter 7, we propose analyses relative to address data flow (*i.e.*, coupling and allocator retrieving). Each of these four analyses is based on one execution, and is a two-steps analysis: an online step during which we perform an instrumented execution to collect data, and an offline step to retrieve the targeted information from the collected data. For each analysis, we detail, in particular, the heuristics we use and the dedicated algorithms. In a third part, we present an implementation of our approach, including technical details about how we perform instrumentation and what actual data we collect. This implementation is presented in Chapter 8. In addition, we propose in Chapter 9 detailed experiments, including a description of our benchmark, and numerical as well as graphical results, to evaluate the accuracy, the accuracy and the scalability of our approach, plus the influence of some parameters and the variability of the results depending on the input of the program under analysis.

8

Contents

Part I

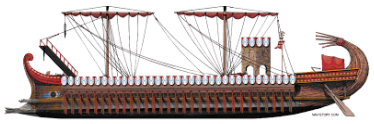
The art of reverse-engineering

Chapter 1

Introduction

In 260 B.C., the Roman Empire was at war with Carthage. Whereas Rome was notably stronger than Carthage on land, they were completely dominated over the sea, as they had no idea how to construct robust galleys. One day, though, they managed to capture one of their opponents boat (see Figure 1.1a), during an apparently insignificant battle in Messina. What this battle changed, is that it gave Rome a model to learn how to construct competitive galleys. They studied the way the Carthage's was constructed, and from it they designed a new version, keeping the strong points of the model and improving the weak points. They also added innovation, such as the famous *corvus* (Figure 1.1b). Within a few weeks, Rome constructed a fleet of such quality that they defeated several times the sea force of Carthage, in particular during the battle of Mylae [Pol70].

(a) Carthage's famous cataphracte



(b) Rome's quadrireme with corvus, largely inspired from the cataphracte

Figure 1.1: The oldest known example of reverse-engineering in history

This is one old example of reverse-engineering that one can find in the literature.

1.1 Instantiation process and reverse-engineering

Reverse-engineering things, as it is illustrated by Polybius' story, consists in inverting what we call the *instantiation* process. The instantiation process is to go, for example, from a technical plan or a recipe to a *product*, a yellow submarine or a great gratin dauphinois. In other words, it is a concretization of a construction process. The product of this instantiation process can either be physical or immaterial: the production of music from a score is also an example of instantiation. This process can be seen as a conversion from ideas and/or information into products, adding some construction information. From an abstract point of view, there is a loss of general information doing this, as we specify it into concrete instances; but on the other hand this concretization leads to the addition of contextual information and construction-relative information that are not described in the instantiation process. For instance, from a blueprint to a building, there is a significant loss of information needed to reproduce the process: the construction of another similar building would be easier from the blueprint than from the final product. However, other information has been added, and in particular methods and process to construct the building from the blueprint. Figure 1.2 illustrates this: the more specific the product goes, the less information it contains in itself, but the more construction-relative information have been added.

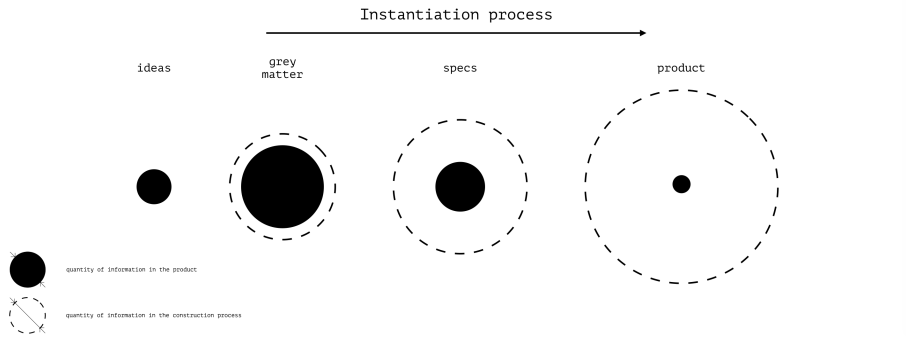


Figure 1.2: The process of instantiation leads to a loss of general information, but adds contextual information

The principle of the reverse-engineering is to retrieve, from the product, part of the information that has been lost during the instantiation process, *i.e.*, a part of the black circle in Figure 1.2 at an earlier stage (often specifications). It is a difficult task, in general, for two main reasons. First, the instantiation process is not reversible: multiple choices can lead to a similar product. Second, this process as well as the construction process can be obfuscated to make the task deliberately harder. Sometimes, however, the instantiation process corresponding to a product one tries to revert is imaginary or has never existed. For instance, in art, the final products (writings, paintings) are hard to express as a product of an instantiation process, and even with good will, authors and painters would probably not be able to describe a method detailed enough to produce new products of similar quality.

1.2 Motivations

1.2.1 Use cases

In the majority of cases, reverse-engineering is needed when the earlier steps of the instantiation process are *not available*. This can be due to several reasons that we detail in the next section. Indeed, the advantages of reversing when the instantiation process is fully known and when one can access any step of it is limited. As we defined *reverse-engineering*, it is trying to get back information lost during the instantiation process. If all the information needed through the chain of this process is accessible, then reversing it has a doubtful benefit. *To summarize, reverse-engineering makes sense when the only thing which is available is the product.* The next question we address is *in what kind of situation is there a lack of information relative to this instantiation process?* We distinguish three scenarios.

1.2.1.1 Information is *not available anymore*.

Plans of a product instantiates ages ago have been lost ; the designer/engineer/creator has left or is dead : etc. In these cases, reversing has, as a goal, to retrieve information that has been definitively lost.

1.2.1.2 Information is *not available*.

One is not qualified to have access to it: a consumer, a concurrent corporation, etc. Here, information exists but is not available to the one studying the product. The goal, in this case, is to retrieve information one is not allowed to access by another way.

1.2.1.3 Information *has never existed*.

The example of the author writing poetry, mentioned earlier, is one of the cases when, by reversing, one is trying to retrieve information that may have never existed as it. Another, more practical, example would be to reverse-engineer a product to understand an unpredicted, an unwanted or an unpurposed behavior and its causes.

These three cases show that situations when doing reverse-engineering makes sense exist and even are common in many fields.

1.2.2 Goals

The main motivation, that can be declined into several sub-motivations, is to understand how the product works and/or behaves internally. This corresponds to gain technical specifications from an instance of one product. Sub-motivations are multiple, we try here to present the most usual ones ; though, this list is not exhaustive.

- **Find out/retrieve functionalities:** from a product for which no documentation is available, find how to use it and what it is capable of.
- **Duplicate/reproduce:** be able to produce another instance of a given product that one have.
- **Modification:** be able to add internal modifications that require a (deep) knowledge of the inside of the product. It can also be to add missing functionalities.
- **Maintenance:** typically repair a dysfunction, a deprecation or a bug.
- **Integration:** be able to interface the product with another system.

- **Testing for safety:** what kind of entries are acceptable by the system, and what are its specifications. In other words, what is the correct domain for testing the product.
- **Testing for security:** also testing, but with a different motivation. The aim here is to study the product from an attacker point of view, and try to compromise it. In this case, reversing is particularly interesting, as it is very close to what a real attacker would do: without information, one does not know *a priori* anything more than an attacker.

1.3 Reverse-engineering computer programs

In computer science, reverse-engineering techniques have been studied in multiple contexts: retrieving protocols from observable exchanges (see [CBP+11], [HGR11]), inferring grammars from queries to an automaton (see [Moo56], [Ang87], [HSJC12]), etc. The context we focus on, in this work, is reverse-engineering computer programs. We present here the specificities of this process when the products are computer programs.

1.3.1 Motivations in computer science

There are two major motivations for reverse-engineering in the context of computer programs: reverse-engineering for maintenance and reverse-engineering for security.

1.3.1.1 Maintenance

For maintenance, there are two main reasons to reverse a program. These motivations have already been presented in Section 1.2, but they are even more important in computer sciences.

- **integration:** in particular, component reuse in another context is difficult if there is no documentation specific to this component: one needs to retrieve format of the inputs, specifications and format of the outputs for instance. This is addressed in [Sha08] and [KSZ+14] for example.
- **testing:** testing a component or a program can be done in black box (see [Gut] and [HGOR13] for instance), but one can use more knowledge on the design of the system under test to produce more efficient tests.

1.3.1.2 Security

From a security point of view, we consider the three main applications to reverse-engineering:

- **consistency:** check anomalies between what we could infer a program should do and what it actually does. For instance, finding that an offline text editor opens a socket and sends files to an IP address located in Russia could be a consistency warning.
- **vulnerability:** analyzing a program can be useful to find vulnerabilities, *i.e.*, bugs that could be exploited to produce a behavior that jeopardize the security of the system. For instance, if the length of a user input is not checked in a program compiled from `C`, it could allow an attacker to execute arbitrary code. For instance, [RM12] and [CGMN12] present analyses of binary codes to detect respectively buffer overflow and use-after-free vulnerabilities.
- **malware analysis:** a reverse-engineering analysis can be performed to decide if a given program is legitimate or is a malware; and in the later case to find countermeasures in order to prevent the execution/propagation of the malicious code (see [BMKK06], [AT15] and [CJS+05] for example).

1.3.2 Instantiation process overview

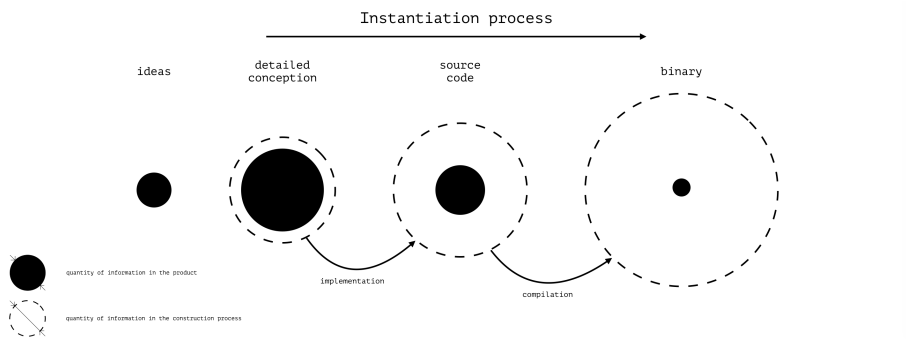


Figure 1.3: Instantiation process of a computer program - main steps

Figure 1.3 presents an overview of the instantiation process in the case of computer programs. This is the instantiation process we target in this work.

1.3.3 Reversing the instantiation process

Reversing a program, in general, aims to recover specifications or behavioral information. We distinguish two possible starting points for analysis, leading to two different points of view. The distinction depends on what one considers as the product he is trying to reverse. In the instantiation process of a program, there are two main steps: implementation of specifications and compilation of the implementation. Thus, either one can consider the source code as the product of the implementation process, or the binary program as the product of the complete chain. These two approaches are very different. Usually, we consider that the amount of information available from the source code is greater than from the binary, which makes sense as it comes earlier in the instantiation process (which leads to a loss of information). Indeed, from the source code, one can recompile the program (and thus rebuild a binary). However, the compilation process may include arbitrary decisions made by compilers, due to some unspecified behaviors according to the specifications of the programming language. This is particularly true in `C` (see [BR10]). Consequently, the same source code, compiled twice with two different compilers, even with the same compilation flags and for the same target architecture, may behave differently at runtime. This point illustrates the fact that these two approaches are to distinguish. The product under analysis is different, and one (the source code) should not be seen as an overlap of the other (the binary) in terms of information. In some cases, a binary will be preferred over source code for analysis. This is the case, for instance, in domains where the safety of the final product cannot be doubted (civil aviation, etc.). It is essential to test the binary program produced by the compiler, as well as the source code and the compiler itself. However, in general, if both source code and binary program are available, one will rather work with source code. This is because, despite the *What You See Is Not What You eXecute* [BR10] phenomenon mentioned earlier, source code contains much more human-readable information than its compiled version, even if some context information (such as concrete addresses, offsets, etc.) are only available at binary level. This means that the main difference between the two approaches is the amount of information available for the analyst, but neither the aims nor the motivations differ significantly. This point being detailed, we present in the next two sections some works relatively to reverse-engineering over source code and binaries.

A few words about obfuscation In addition to the complexity of reverse-engineering due to the lack of information about the construction process, sometimes effort is made by the creators of a given product to make the task even harder. It is called *obfuscation* and can be performed either at source level or at binary level. The principle is to deliberately hide relevant information, either with noise (*e.g.*, opaque predicates - see [XMW16]) or with additional complexity (*e.g.*, packers - see [GMS+15]). Several works address this issue and propose techniques to defeat obfuscation (see [EGV16] and [BRTV16] for instance). However, we do not focus, in this work, on obfuscation and deobfuscation, and most of the time we assume that no active effort was made to make the reverse-engineering process particularity harder.

1.3.3.1 From source code or specifications

From source code, one doing reverse-engineering typically targets to reconstruct a design-level abstraction for several applications. For instance, in [RILR14], the authors present a method to analyze the design of an application with some security meta-data, and prove that the system design is compliant with some security specifications. In particular, they aim to show that it is impossible by design (but this does not mean that an implementation vulnerability cannot occur) to reach a given state without authentication. This work and many others (for instance, [NYH+15]) apply to a design-level representation. That is why [KSRP99], [SO06] and more recently [WPV17] propose methods to reconstruct higher-level information from source code. Another common reason to perform source-code reversing is to test, component by component.

1.3.3.2 From binary

To perform a reverse-engineering analysis of a binary, there are two main approaches, described in the following paragraphs: a linear approach, where one tries to revert the instantiation process step by step, and a direct approach where one tries to get relevant information directly (with no intermediate step). The difference between the two, in the context of binaries, is that the first requires to revert the compilation process: it is called decompilation.

Decompilation Decompilation consists in retrieving, from the binary under analysis, a source code from a higher-level language that corresponds to the program,

and then perform the analysis on the result as if it was the original source code. This process is called *decompilation*. The first step of decompilation is usually to translate the binary to assembly code (*e.g.*, `x86-64` or `ST20`). This is called *to disassemble*. In general, it is commonly accepted that this step is easy. Indeed, going from assembly to binary is almost a literal translation in most cases. Therefore the process can be reversed with a good accuracy. This step is discussed more precisely in Section 2.3. Several open-source tools are able to disassemble almost any binary (`objdump` [Fouc], `miasm` [Des], etc.). In some particular cases, however, it can be a difficult problem. In the scope of this work, we consider that this step is possible (and this is true for any legitimate binary produced by a compiler). Then, from the assembly code, comes the decompilation step: try to retrieve a source-level code (for instance in `C`) which would lead to a similar assembly code if re-compiled. This step is much harder than to disassemble, as the loss of information during the instantiation process at this step is considerable. Some work has been conducted in this area ([CG95], [PW97]) but the source code produced is often unnatural, hard to understand and over-complicated. Thus it is of little help, and in facts it is often less readable than the assembly code. Within the last ten years, very few papers have been written on decompilation (for instance, [DG11] is about decompilation of Android applications), which let one suppose that this is not the dominant approach nowadays.

Direct approach The direct approach consists in reversing the two steps of the instantiation at once (compilation and implementation), and do not consider the source code as a step in the reversing process. The aim is to retrieve behavioral and/or specification-level information directly from the binary. When doing a semantic analysis from the binary directly, we can once again distinguish two types of analyses:

- a static analysis, which consists usually in disassembling and then analyzing the semantics of the assembly language (see Section 3.1),
- a dynamic analysis, which consists in executing the binary program, either with or without some instrumentation, and observe its responses. Responses can be outputs as well as any kind of observable data (time of execution, memory use, register values, etc.). See Section 3.2.

In most cases, analyzing a binary program is a mix of these two approaches: to disassemble to get assembly code is a very frequent step in both static and dynamic

analyses. In this work, we perform dynamic analysis, but our instrumentation requires a step of disassembling as well. In the next chapter, we present more specifically the different techniques for semantic analysis of programs at binary level (with no source code).

Chapter 2

Reverse-engineering binaries

In this chapter, we focus on binary analysis, with the aim to retrieve high-level information. We describe in Section 2.1 the specificities of working on binaries rather than at source-level, and Section 2.2 gives an overview of the existing analyses on binaries. Examples we provide in each section of this chapter are given in `x86(-64)`. In addition, note that in this section and in the rest of this work, we use equally the 32bit or the 64bit notation of registers (*e.g.*, we use `%eax` as well as `%rax`).

2.1 Specificities of working on binaries

Working at assembly/binary level presents some particularities that make the analysis (very) different from what we can do at source level. Other problems appear, and difficulties inherent to the lack of high-level information require specific attention. These particularities are well described in [BR10]. The main ones are:

- **no notion of type** - Hardware-stored values are not typed, they only consist in a sequence of bits, whereas variables in high-level languages are usually typed, which allows to perform type-checking and other consistency analyses. Types also provide information about the way data may be handled.
- **registers** - A part of the computation at assembly level is handled in registers, which makes the data flow analysis more complicated to handle than it is at source level with decorators (variable names).

- **indirect addressing and indirect branching** - This mechanism consists in using a dynamic value (*e.g.*, the content of a register) as an address value to perform either a branching or a memory access. It makes the control flow and the data flow hard to analyze, because of the dynamic computation of addresses. For instance, `%ebp + 4` could target the same memory location as `esp`, whereas it is not explicit at binary level.

2.2 Learning from a binary

In this section, we present an overview of **what** information that can be seen as *to be retrieved* from binaries, and **why** they could be interesting. We distinguish three categories of information targeted by binary analyses:

- **structure**: information relative to the design of a program. Most of the time, works suppose the binary is produced by a compilation process, and try to recover source-level information lost during this process.
- **control-flow**: information relative to the execution of the program. Influence of inputs, sequences of execution, etc.
- **data-flow**: information relative to data propagation.

The next section will present common techniques used to perform these analyses. We can distinguish four levels of analysis of a program: **black-box**, **binary**, **assembly** and **source**. In some cases, working on a binary itself is legitimate, but generally, the first step of binary analysis is to disassemble it. The following subsection presents the aims, approaches and difficulties relative to disassembling a binary. We consider then that binaries can be disassembled correctly, and we use equally both binary (to execute code) and assembly (to instrument, etc.) representations, assuming they are equivalent.

2.3 From binary to assembly

Almost any static analysis and many dynamic instrumentation techniques on a binary start by disassembling it.

2.3.1 Disassembling

We denote here, by *disassembling*, the process consisting in going **from a sequence of bytes** (*i.e.*, the part of the binary program which is executable) **to assembly-level instructions**. In some other contexts, *disassembling* a binary could mean more than that (for instance, it could include basic blocks reconstruction - see relative section below), but we exclude this from the scope of our work.

2.3.2 Issues

One of main issues when disassembling instructions from a binary is to **differentiate code from data**. Indeed, given a sequence of bytes corresponding to instructions, it is straight-forward to recover the corresponding assembly code. For instance, `0x06` corresponds to the instruction `PUSH` in `x86`. However, knowing what bytes are to be disassembled and what bytes correspond to data is a difficult task. With `x86` assembly, another difficulty is to be handled: instructions have a variable size. Some instructions are of one-byte length, and others can be of up-to fifteen bytes. In other assembly languages, for instance `MIPS` or `ARM`, instructions have a fixed size. This means that the result of decoding strongly depends on the offset it starts. In `ARM`, instructions are four-bytes long. Given a sequence of bytes, there are only four possible ways to decode it, depending on the offset modulus 4 at which one starts to disassemble. Because of variable-size instructions, this assumption is not true in `x86`: there are much more ways to decode a sequence of bytes, depending on the starting offset.

2.3.3 Approaches

We present here the two main categories of approaches to disassemble a binary program. These approaches have been improved by several works ; however limitations we detail here still stand.

2.3.3.1 Linear sweep

The most simple strategy to disassemble a program is to start from the beginning of the code section (often given in the section header table, see Listing 2.1 for example), and assuming that this section only contains code. Disassembling is performed sequentially, byte by byte.

- First, the opcode is read (it codes the instruction to be executed) – note that in some cases, the opcode can by encoded with more than one byte (in this case, the first byte is always `0x0F` in `x86` for instance).

- Depending on the opcode, zero, one or several bytes are read, corresponding to the operands of the instruction being decoded. For instance, the instruction encoded by `0xFF` takes one operand of one byte.

- This is performed until some opcode decoding fails or the end of the code section is reached.

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400200  00000200
       000000000000001c  0000000000000000   A       0     0     1

  [...]

  [13] .text             PROGBITS         00000000004003c0  000003c0
       00000000000001a2  0000000000000000  AX       0     0     16

  [...]

  [29] .strtab           STRTAB           0000000000000000  00001780
       0000000000000226  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Listing 2.1: Example of a ELF section header table - obtained using the command-line tool `readelf`

This algorithm (or an improved form of it) is used by famous open-source tools (`objdump` [Fouc], `WinDBG` [Cor]). However, when there are bytes of data in the middle of some code, this technique is often unable to output correct assembly code. Consider the example, inspired from [Luk], given in Listing 2.2.

```
int main(void) {
    asm (
    "A:\n"
        "movq $A, %rax\n"
        "addq $15, %rax\n"
        "jmpq *%rax\n"
    );
    asm volatile (".byte 0xab");
    asm volatile (".byte 0xcd");
    asm (
        "movl $17, %eax\n"
    );
    return;
}
```

Listing 2.2: Example of source code that defects a linear sweep strategy for disassembling

This `C` code use the `asm` keyword to include `x86` instructions, and `asm volatile` to include data bytes. In this code, we intentionally include some data bytes in the middle of assembly code. The first three lines of assembly aim to jump to the instruction `movl $17, %eax`. During the execution, the data bytes are thus not executed. However, a linear sweep disassembling will output an inconsistent, for the following reasons:

- the data bytes `0xAB` and `0xCD` will be considered as code and will be disassembled ;

- instruction `0xAB` (`stos %eax,%es:(%rdi)`) has no operand, and `0xCD` has one operand of one byte. This means that the next byte, corresponding to the instruction `movl`, will be considered as an operand of `0xCD`. From there, all disassembled instructions that follow are incorrect, because data to disassemble is no longer aligned with opcodes.

Listing 2.3 presents the result of disassembling for this program, obtained with `objdump -d`.

```
00000000004004ad <main>:
  4004ad:       55                      push   %rbp
  4004ae:       48 89 e5                mov    %rsp,%rbp

00000000004004b1 <A>:
  4004b1:       48 c7 c0 b1 04 40 00    mov    $0x4004b1,%rax
  4004b8:       48 83 c0 0f             add    $0xf,%rax
  4004bc:       ff e0                   jmpq   *%rax
  4004be:       ab                      stos   %eax,%es:(%rdi)
  4004bf:       cd b8                   int    $0xb8
  4004c1:       11 00                   adc    %eax,(%rax)
  4004c3:       00 00                   add    %al,(%rax)
  4004c5:       90                      nop
  4004c6:       5d                      pop    %rbp
  4004c7:       c3                      retq
  4004c8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  4004cf:       00
```

Listing 2.3: Result of disassembling obtained with a linear sweep algorithm on the code presented in Listing 2.2

Although this example was manually constructed to illustrate the issue of disassembling, it does happen in practice with common programs (see [MM16]).

2.3.3.2 Recursive traversal

A second approach, implemented in particular by `IDA Pro` [HR] and `OllyDBG` [Yus], consists in following statically, and as exhaustively as possible, the control flow, to disassemble every reachable code, and to avoid disassembling data bytes. The principle, taken from [SDA02], is given in Algorithm 1. The example given in Listing 2.2 is correctly disassembled by this method, with the assumption that a disassembler is able to evaluate statically the target of the `JMPQ` instruction. However, with more complicated indirect `JMP` or `CALL`, it becomes difficult to follow the control flow with a static approach (for instance when functions are called through a pointer dynamically computed), and this method is not able to disassemble blocks of instructions no `JMP` (or equivalent) instruction targets explicitly. .

2.3.4 Assumption

Despite the limitations of the two classical approaches for disassembling (linear sweep and recursive traversal), in most cases, disassembling will give accurate

```
Procedure RecDisassemble(addr, instrList)
    Input: addr: address of the instruction to disassemble, instrList: a list
           of instructions already disassembled
    begin
        if addr has already been visited then
            return
        end
        repeat
            instr ← DecodeInstr(addr);
            addr.visited ← true;
            instrList ← instrList ⋃ {instr};
            if instr is a branch or a function call then
                T ← set of possible control flow successors of instr;
                foreach target in T do
                    RecDisassemble(target, instrList);
                end
            else
                addr ← addr + instr.length;
            end
        until addr is not a valid instruction address;
    end
```

Algorithm 1: Recursive Traversal Disassembling Algorithm

results (*i.e.*, assembly code close to the original one). In the scope of this work, we do not consider problems due to disassembling, and consider that working on a binary and working on its assembly code is equivalent. In other terms, we consider that **the assembly step is reversible**.

2.4 Structure information

Although we mentioned in Section 1.3.3.2 that decompilation is not the way usually chosen, because of its hardness and the quality of the source code produced, one can still want to retrieve some kind of information available at source level but that has been lost during compilation. Note that in this scenario, the aim is not to reproduce a complete source code from which it would be possible to apply source level analysis techniques ; instead we target global information for general knowledge. What we call *structure* in this section, is between syntactical-level and semantic-level. It is not syntactical, because we do not aim to retrieve, for instance, function prototype the exact way they were expressed at source-level. On the other hand, it is not either strictly semantic, because we do consider sizes and types of variables for instance.

2.4.1 Functions

Functions[1] are a **powerful tool to describe** objects, processes, schemes, etc. In programming, it is the key element to structure algorithms, just as data structures; and almost every language is based on functions, from `C` to `Python`. For these two reasons (powerful to describe and basis of programming), retrieving functions from a binary is the topic of many research works (see [BBW+14] and [ASB17] for instance). Most of the work on this topic, that we present in the next paragraph, suppose that the binary program is obtained by compilation from source code (often from `C`[2]). In these examples, what is meant by function is therefore a source-level equivalent of a compiled function embedded in a binary. In fact, it is almost exactly the process of decompilation which is (partially) targeted. In Part II, we propose a general definition of functions at binary-level, but for now we keep the meaning of function that we have at source-level.

[1] In the meaning of subpart of a program/subroutine

[2] Although other languages would produce different types of binaries, for instance object-oriented languages

2.4.1.1 Location

At source level, functions are usually well delimited. This delimitation is lost during compilation, and optimizations make it even harder to retrieve. In assembly code, the main clues one can use to retrieve functions are:

- `CALL` instructions to find entry points ; e.g. `call 0xCAFE`,
- `RET` instructions to delimit the end of functions.

However, multiple points induce complexity, and make the problem of location of functions in a binary hard to solve:

- multiple `RET` - see Listing 2.4: one function can return from different `RET` instructions, for example if there are `RET` instructions in different branches;
- no `RET` - see Listing 2.5: a function can either never return or return with another instruction than `RET`;
- multiple entry points: a function execution can start from different instructions;
- dynamic `CALL` - see Listing 2.6: the target of a `CALL` is computed dynamically, and thus cannot be inferred statically;
- inlining of functions: each call is replaced by the actual code of the function.

```
0000000000400550 <f>:
400550:       53                      push   %rbx
400551:       89 fb                   mov    %edi,%ebx
400553:       e8 b8 fe ff ff          callq  400410 <rand@plt>
400558:       ba 56 55 55 55          mov    $0x55555556,%edx
40055d:       89 c1                   mov    %eax,%ecx
40055f:       f7 ea                   imul   %edx
400561:       89 c8                   mov    %ecx,%eax
400563:       c1 f8 1f                sar    $0x1f,%eax
400566:       29 c2                   sub    %eax,%edx
400568:       8d 04 52                lea    (%rdx,%rdx,2),%eax
40056b:       29 c1                   sub    %eax,%ecx
40056d:       83 f9 01                cmp    $0x1,%ecx
400570:       74 26                   je     400598 <f+0x48>
400572:       83 f9 02                cmp    $0x2,%ecx
400575:       89 d8                   mov    %ebx,%eax
400577:       74 09                   je     400582 <f+0x32>
400579:       85 c9                   test   %ecx,%ecx
40057b:       74 0b                   je     400588 <f+0x38>
40057d:       b8 ff ff ff ff          mov    $0xffffffff,%eax
400582:       5b                      pop    %rbx
400583: ----  c3 --------------------  retq
400584:       0f 1f 40 00             nopl   0x0(%rax)
400588:       89 df                   mov    %ebx,%edi
40058a:       c1 ef 1f                shr    $0x1f,%edi
40058d:       01 f8                   add    %edi,%eax
40058f:       d1 f8                   sar    %eax
400591:       5b                      pop    %rbx
400592: ----- c3 --------------------  retq
400593:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
400598:       8d 44 5b 01             lea    0x1(%rbx,%rbx,2),%eax
40059c:       5b                      pop    %rbx
40059d: ----- c3 --------------------  retq
40059e:       66 90                   xchg   %ax,%ax
```

Listing 2.4: Example of a function with multiple RET instructions

```
00000000004005a0 <g>:
4005a0:       48 83 ec 08             sub    $0x8,%rsp
4005a4:       e8 67 fe ff ff          callq  400410 <rand@plt>
4005a9:       48 83 c4 08             add    $0x8,%rsp
4005ad:       89 c7                   mov    %eax,%edi
4005af:       eb 9f                   jmp    400550 <f>
4005b1:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
4005b8:       00 00 00
4005bb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

Listing 2.5: Example of a function with no RET instruction

```
400609:       41 ff 14 dc             callq  *(%r12,%rbx,8)
```

Listing 2.6: Example of an indirect `CALL` instruction

This problem of retrieving functions is addressed in [CJMS10] for example, and several tools propose solutions (IDA [HR], etc.) based on heuristics. More recently, [ASB17] addresses specifically this problem, and present a static analysis that shows very accurate results, when comparing the functions retrieved from the binary with the functions declared at source level.

2.4.1.2 Interface

For code reuse, fuzzing, testing, integration, understanding, etc., it is important to get the interface of functions, and in particular information about its inputs/outputs (i/o). Listing i/o is a complex task, and it begins with defining what should be considered as input and what should not. For instance (see Listing 2.7), should a global variable (here `NB_ITER`) read by a function `f` but not given as a parameter through registers or through the stack should be considered as a parameter of `f`?

```
const int NB_ITER = 7;

int f(int a) {
    int i, res = a;
    for (i = 0; i < NB_ITER; i++)
        res = g(res);
    return res;
}
```

Listing 2.7: Function using a global variable `NB_ITER` as an `in` parameter

If one aims to get a source-level description of the program as accurate as possible, the answer should be no. On the other hand, in [CJMS10], the goal pursued by retrieving function interfaces is to extract from the binary every byte needed to be able to execute it in another context. In this case, `f` depends on a global variable, so it should be seen as an input to extract. We propose in Chapter II our own definition of inputs and outputs at binary level. In [FZPZ08], Wen Fu et al. specifically target the particular case of variable-argument functions: how to distinguish them and how to recover the variable arguments. In addition to the number of i/o, the nature of their value is a valuable information. Types are

usually described at source level with an explicit declaration, in the case of compiled languages (`C`, `Ada`). This is useful, in particular, to proceed to type checking at compilation time. However, this meta-data regarding variables is removed then and is not present at assembly/binary level. Several works aim to recover the type of parameters of a function. In [BBW+14], the authors present a static analysis to recover function prototypes from binary, including types of parameters. Usually, the set of types being considered is a restriction of the `C` types. For instance, size of integers is not always targeted as an important information.

2.4.2 Variables

Functions are relative to control-flow. Regarding data-flow, a program manipulates data to handle function inputs, outputs, intermediate computations, etc. Data is used to represent information needed during the execution (either to produce a result or to drive the control-flow). These data are characterized by the nature of the information they represent ; this nature defines the operations that are computable on it, and vice-versa. It also defines the nature of new information produced from one or several ones (by computation for example). Usually, this nature is expressed by a **type** at source-level. Data are stored in memory, at a specific **location** (file, memory address, etc.). At source-level, the notion of *variable* is used to link this location to an identifier, and thus abstract the notion of address. To our knowledge, except for very specific functional languages such as `Unlambda`, *variables* exist in every high-level language. For these two same reasons we already presented for function inference, many works have focused on variable recovery from binary (see [BR07], [EAK+13] for instance). As for function recovery, works usually assume that the binary under analysis is produced by a compiler, and thus try to retrieve variables in the meaning given at source-level. In Chapter 5, we also give a general definition of a variable (*i.e.*, of memory locations, types and sizes) at binary level.

We encounter issues with variable that are very similar to the ones we described relatively to functions in the previous section.

2.4.2.1 Location

A variable name at source-level corresponds to a memory location in assembly (or at binary level). This memory location can be in the stack, in the heap, in the data section, etc. of the binary. One of the problems at this level is to identify

them. Local variables of functions, for instance, are usually in the stack, and accessed relatively to the base (or the top) pointer of the stack (according to some compilation conventions - in the case they are not respected, this is even harder). However, the number of local variables is not known a priori. Let us take an example: if eight bytes on the stack are allocated for local variables, it could be for eight one-byte integers (8 variables), four two-bytes integers (4 variables), or four-bytes integers (2 variables). One thus needs to analyze instructions of the function to find how this range of bytes in the stack is handled. Figure 2.1 illustrates this.

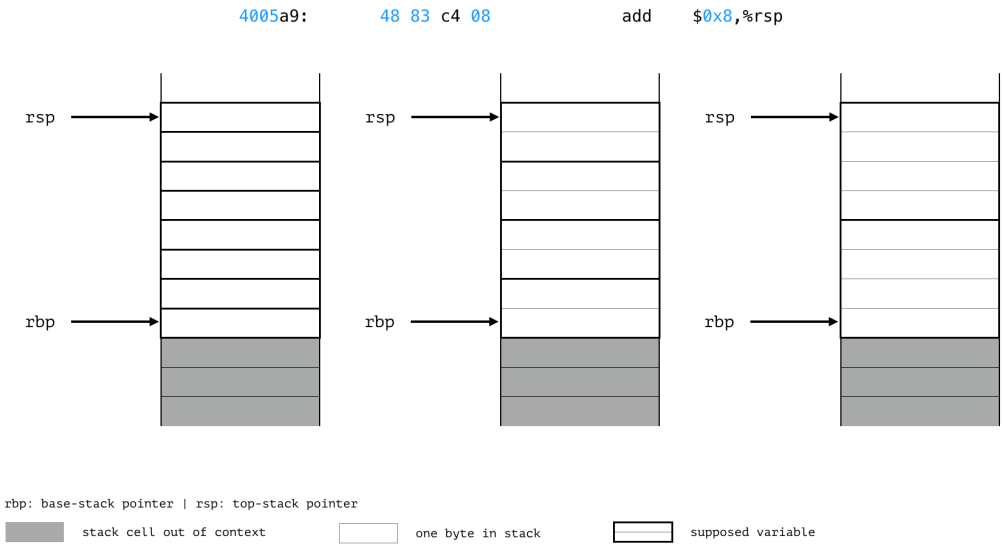


Figure 2.1: Stack frame for local variables

2.4.2.2 Type

A main difference between assembly/binary code and other languages, as mentioned in Section 2.1, is the total absence of meta-data relative to the type of variables. In an assembly code obtained from source code by compilation, variables are semantically typed (they were [almost always] typed at source level, and a compiler

usually performs type checking to ensure that operations are compatible with the nature of variables). However, their types are not explicit anymore and need to be retrieved in order to have a better understanding of the role of each one. To retrieve types, one possibility is to do it from the operations performed over variables. In other words, at source-level the type of a variable determines the operations that can be performed with it, whereas at assembly level we use the operations to determine the type. Multiplication, for example, would not be performed between arrays and floats. Another approach would be to base the type recovery on concrete values of variables: small values (between 0 and 2^{16} for instance) are likely to be integers whereas values between 2^{48} and 2^{64} are likely to be addresses (on a 64-bit architecture).

2.4.2.3 Size

Another meta-data that we sometimes have at source level, but that is most often a compiler's choice, and that is lost during compilation is the size of variables in bytes. Usually, variables are stored in four or eight bytes memory locations (to maintain memory alignment) and operations are also performed on four (on 32 bit architectures) or eight (on 64 bit architectures) bytes in registers for optimization. Thus, by opposition to the problem of types where operations determine the type of the variable, it often leads to no information on the size. However, this size is meaningful although most of operations are performed on all the bits of a register, only some of those bits have a significance and are useful semantically.

2.4.3 Data structures

> *"Forensics and reverse-engineering without data structures is exceedingly hard. [...] Since real programs tend to revolve around their data structures, ignorance of these structures makes the already complex task of reverse engineering even slower and more painful."* [SSB11]

Data structures (in the meaning of C-like structures with several fields) can be seen as a particular type of variable, which consists in an aggregation of several variables of various sizes and types. Recovering data structures from a binary is a subject of research works, and presents several difficulties (see [SSB11]):

- To distinguish structures from single variables. Addressing `reg + 4` can be an access to a variable on the stack for instance (if `reg` is `%ebp` or `%esp`), or it can be an access, from a base pointer, to a field of a structure.

- To distinguish structures from arrays. `reg + 4` can also be an access to a particular cell of an array. This distinction is harder to make, according to [SSB11]. More details are given in Section 3.2.

- To distinguish high-level data structures (in the algorithm meaning). Usually, C-like structures are used to implement data structures such as binary trees, linked lists, etc. To retrieve them presents an additional level of complexity: it requires to retrieve, in addition to the fields of a structure, their semantics.

2.4.4 Object-Oriented Programs (OOP)

Object-oriented programming (OOP) comes with particular schemes that can be useful to recover as well to better understand a program. In OOP, an object has *attributes* and methods, and it is those fields that we are usually interested in retrieving. Both can be either specific to the object's class or inherited from a parent class. This is an additional difficulty in recovery. Others may occur as well: for instance, for each attribute, a distinction has to be made between a class attribute (shared by every instance of this class) and an object attribute (specific to one instance). In [JCG+14], authors propose a technique to retrieve C++ object attributes from the compiled program, by tracking accesses performed from the *this* pointer.

2.5 Control-flow

In addition to source-level information, which is mostly static, reversing also aims to retrieve dynamic patterns, and especially on the way and the order instructions are executed. This is called *Control Flow*. Note that the *Control Flow* analysis is not specific to binaries, but can also be relevant at source-level. This section presents several levels of analysis relatively to *Control Flow*.

2.5.1 Basic blocks

2.5.1.1 Presentation

A basic block is a sequence of instructions that will be executed sequentially, from the program point of view. Basic blocks are ended by an instruction that will interfere with the control flow, such as a function call or a (conditional) jump.

Such blocks are useful for the understanding of a program, as it divides it in small bricks that can be analyzed (almost) independently from the rest of the program.

2.5.1.2 Main difficulties

Retrieving the end of basic blocks from assembly code is, in most cases (*i.e.*, from most compiled binaries), straight-forward: instructions that change the control flow are well-identified (*e.g.*, `CALL`, `JMP`, `RET`, etc.), and mark a discontinuity in the execution. The beginning of blocks is harder to retrieve: a basic block begins at the first instruction that will be executed at runtime after the end of a previous basic block. This means that the beginning of a basic block is the target of the end of another basic block. An instruction `JMP 0xCAFE` at the end of the basic block `A` means that another basic block begins at `0xCAFE`. But the target of branching instructions can be hard to determine statically, and some basic blocks can be hard to activate (*i.e.* find a trace that will lead to execute it) dynamically. The second difficulty is when basic blocks are obfuscated, for instance using opaque predicates.

2.5.2 Control Flow Graph (CFG)

From the basic blocks, retrieving a CFG consists in linking basic blocks from one to another. In other words, it requires to retrieve which basic block is reachable from which basic block. The CFG is useful to better understand the possible paths an execution can take.

2.5.3 Call Graph

At function level, it is valuable to understand how the blocks (*i.e.*, functions in this case) interact with each other; A part of this work is called *Call Graph* retrieving. A *Call Graph* describes interprocedure calls. A typical representation is an oriented graph, where an edge from f to g means that a call to g can occur during the execution of f.

Remarks.

1. *Call Graph* is a function-level notion, therefore it only makes sense for programs compiled or written according to the function paradigm. 1. An edge from f to g means a call from f to g can occur, but it does not mean

it will necessarily occur. Indeed, the call can depend on some condition evaluated regarding inputs of f or global variables.

2. The *Call Graph* gives a good picture of the possible interactions at function level, but it is a static picture, and thus some information describing the execution are missing. For instance, consider the simple example given in Figure 2.2.

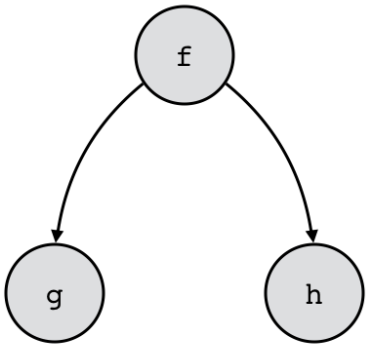


Figure 2.2: Simple example of a call graph where `f` calls `g` and `h`

From this graph, we cannot tell if a call to f leads to a call to g **and** a call to h (and if yes, in which order), or if it is g **or** h, etc.

2.6 Data-flow

In addition to the control flow, the data flow can give interesting information about how the program handles inputs for instance.

2.6.1 Data tracking

User inputs of a program usually have an influence on the control flow, and also on output values. More generally, data impact control flow, and data impact data.

From a functional perspective, parameters often influence the output value, and the output value of *f* can have an impact on the output value and/or the control flow of *g*, *h*, etc. For a given data *d*, forward analyses aim to determine what data *e*, *f*, *g* is influenced by its value, and backward analyses try to retrieve all the data *a*, *b*, *c* that influenced *d*. Figure 2.3 illustrates this example.

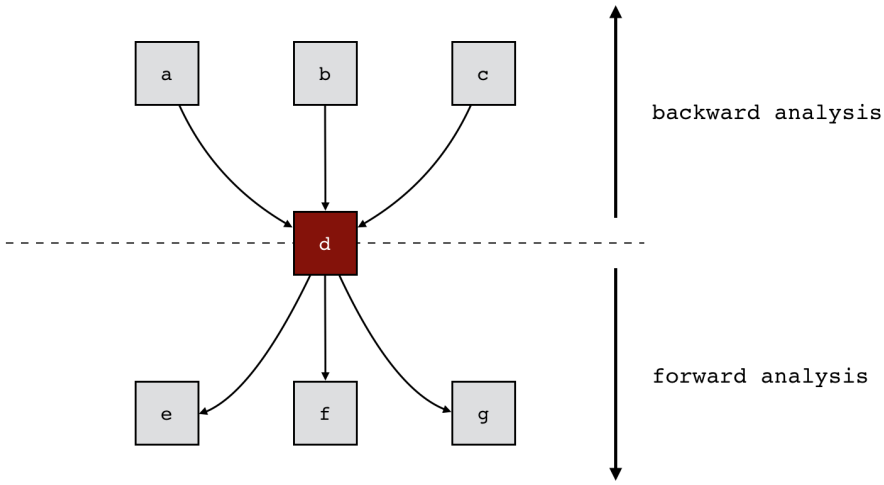


Figure 2.3: Simple example of a data tracking - backward and forward analyses

Forward data tracking is often performed using a taint analysis (see [NS05], [SAB10], [ARF+14]): each data is associated with a color, and each operation using this data propagates the color. The final color (which is a mix of colors) tells which data influenced the result. A typical use case of these analyses is to propagate the user inputs, and confirm that none of the critical functions take in parameter a tainted value (see [NS05]). Data tracking can be performed at several levels of granularity. Some works focus on memory locations, and others study the propagation at bit level. An example is given in Listing 2.8. In this case, the parameter given in register `%edi` influences the return value (`%eax`): `%edi -> %ebx -> %eax`. However, an analysis at bit-level shows that only the least significant bit of `%edi` has an influence on the output.

```
mov     %edi,%ebx
xor     0x1, %ebx
mov     %ebx, %eax
add     %esi, %eax
ret
```

Listing 2.8: Simple example for taint propagation at bit level

2.6.2 Memory access

There are many applications where memory accesses are studied. The main one we will focus on is for safety and security reasons. An example of situation where the memory analysis is required is memory leaks detection (*e.g.*, `Valgrind` [NS07]): a long-term running program with memory leaks will crash eventually by a lack of memory space left. Regarding security, many well-known vulnerabilities on binary are relative to memory accesses (see next section). The specificity of a memory analysis is to know if an access to a given memory location, either to read or to write, is legitimate. And because, among the addressable space, some memory locations are allocated and some are not, the access rules can be complicated. For instance, a memory location that is not allocated should not be accessed, neither to read nor to write, except by an allocator. Knowing what location is allocated and what is not is one of the issues to be solved. Another one is to find if a given content can be overwritten or not.

2.6.3 Vulnerability detection

As mentioned in a previous section, data flow on addresses can be used for security purposes. In particular, many techniques have been studied to detect, from a binary, vulnerabilities. For instance:

- **stack buffer overflow** - see Listing 2.9. Buffer overflows correspond to a write that occurs after the end of an allocated buffer.

```
#include <stdio.h>

int main() {
    char buf[8];
    int admin = 0;
    gets(buf);
    printf("%s\n", buf);
    printf("%i\n", admin);
    return 0;
}
```

Listing 2.9: Example of a simple buffer overflow vulnerability

In Listing 2.9 for example, the function `gets` writes in `buffer` every character given in the standard input by the user, even if there are more than eight. Giving too many characters in the standard input leads to a re-write of the variable `admin`. For instance, in the following command, we give as an input 8 legitimate characters (`'01234567'`) and then 5 characters to overflow the array `buf`:

```
> python -c "print '01234567' + chr(7) * 5" | ./bof
buf: 01234567
admin: 7
```

We see in the output of the program that the value of the variable `admin` has been overwritten by the input. It can be even more critical if a return address (stored in the stack) can be written by the user: it allows an attacker to modify the control flow of the program, and possibly to execute arbitrary code. Canaries [Cow98] and other works [BST00] focus on their detection, because they could lead to major security issues (see Morris Worm [mor] and Slammer Worm [sla] for instance). The idea of canaries is to add random values just before critical information in the stack. These values are then checked before accessing the critical information. A modification of these critical values through a buffer-overflow vulnerability implies to re-write the canary, and because its value is random, it is hard for an attacker to re-write the correct value.

- **heap overflow** - see Listing 2.10 taken from OWASP.

Heap overflow is an overflow similar to what we presented in the previous section, except that it occurs on the heap instead of the stack. In this example, two blocks are allocated, and a large write from the first one (`buf0`) ends up overriding the content of the block pointed by `buf1`. Detecting heap overflows on binaries have been studied in [RKMV03] and [AF11] for instance. From a binary, [Ber06] proposes to add a canary in meta-data relative to each memory block (these meta-data are stored by `malloc` just before the beginning of blocks). When a block is released, `free` will check the consistency of this canary: a mismatch means that this block was corrupted. Note that this is a runtime detection that requires to re-implement `malloc` and `free` routines, but it can be applied to binaries without re-compilation, for instance using the `LD_PRELOAD` trick or by replacing the library file `libc.so` (which is dynamically loaded).

- **use-after-free** - see Listing 2.11 taken from [FMP14]. Use-after-frees occur when a memory location is accessed after it has been freed by the allocator. If this later access is a read, then it can lead to data leak, especially if this location has been reallocated to store sensitive data. In the example given in Listing 2.11, there is a path of the execution where the pointer `p_global` is freed and then accessed (if line 20 is executed, then `p_global` is not restored and freed at line 36).

  [CGMN12], [You15] and [Hee09] for instances propose techniques to detect use-after-free vulnerabilities from the binary code. [FMP14] rely on a static analysis in three steps:

  - track heap operations (including aliases)
  - statically detect vulnerable accesses (on free blocks)
  - extract subgraphes (for each vulnerability), from the allocation, through the free, and to the later access.

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

#define BSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */

void main(void) {
   u_long b_diff;
   char *buf0 = (char*)malloc(BSIZE);      // create two buffers
   char *buf1 = (char*)malloc(BSIZE);

   b_diff = (u_long)buf1 - (u_long)buf0;   // difference between locations
   printf("Initial values:  ");
   printf("buf0=%p, buf1=%p, b_diff=0x%x bytes\n", buf0, buf1, b_diff);

   memset(buf1, 'A', BUFSIZE-1), buf1[BUFSIZE-1] = '\0';
   printf("Before overflow: buf1=%s\n", buf1);

   memset(buf0, 'B', (u_int)(diff + OVERSIZE));
   printf("After overflow:  buf1=%s\n", buf1);
}

[root /tmp]# ./heaptest

Initial values:  buf0=0x9322008, buf1=0x9322020, diff=0xff0 bytes
Before overflow: buf1=AAAAAAAAAAAAAAA
After overflow:  buf1=BBBBBBBBAAAAAAA
```

Listing 2.10: Example of a heap overflow vulnerability

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

#define SECRET_PASS 2784621
#define MIN 0
#define MAX 100

int *p_global;

int cmp() {
    return *p_global >= MIN && *p_global <= MAX;
}

void index_user(int *p) {
    int *p_global_save;
    p_global_save = p_global;
  p_global = p;
    if (cmp() <= 0) {
        printf("The secret is greater than 50\n");
        return;
    }
    printf("The secret is less than 50\n");
    p_global = p_global_save;
    return;
}

int main(void) {
    int *p_index, *p_pass;
    p_global = (int *) malloc(sizeof(int));
    *p_global = SECRET_PASS;

    p_index = (int *) malloc(sizeof(int));
    printf("Give a number between 1 and 100 \n");
    scanf("%d", p_index);
    index_user(p_index);
    free(p_index);

    p_pass = (int *) malloc(sizeof(int));
    printf("%p | %p\n", p_pass, p_global);
    printf("Give the secret\n");
    scanf("%d", p_pass);

    if (*p_pass == *p_global)
        printf("Congrats!\n");
    else
        printf("Sorry...\n");
    return 0;
}
```

Listing 2.11: Example of a use-after-free vulnerability

2.7 Stack Management

2.7.1 Calling Convention

The vast majority of programs uses functions, and therefore, function calls. It can be internal calls or calls to functions embedded in libraries, either statically or dynamically loaded. When a function `f` calls a function `g`, or a function `glib` from a library, both the caller (`f`) and the callee (`g` or `glib`) must *agree* on the way they communicate. In particular, they must agree on the way parameters are passed from the caller to the callee and on the way the stack is cleaned up after the call.

2.7.1.1 Passing parameters

There are two classical ways, at assembly level, to pass parameters:

1. Through the stack: the parameter value will be pushed on the stack by the caller and accessed by the callee. Listing 2.12 shows an example of `f` calling `g` passing a parameter through the stack.

2. Through registers: the parameter value will be written in a given register by the caller and read by the callee. Listing 2.13 shows an example of a parameter being passed through a register.

```
080483cb <g>:
 80483cb:       55                      push   %ebp
 80483cc:       89 e5                   mov    %esp,%ebp
 80483ce:       83 ec 10                sub    $0x10,%esp
 80483d1:       8b 45 08                mov    0x8(%ebp),%eax
   ...:   ...           ...

080483e5 <f>:
   ...:   ...          ...
 80483fa:       ff 75 fc                pushl  -0x4(%ebp)
 80483fd:       e8 c9 ff ff ff          call   80483cb <g>
```

Listing 2.12: Example of parameter passed through the stack

```
00000000004004ad <g>:
  4004ad:       55                      push   %rbp
  4004ae:       48 89 e5                mov    %rsp,%rbp
  4004b1:       89 7d ec                mov    %edi,-0x14(%rbp)
   ...:   ...           ...

00000000004004c8 <f>:
   ...:   ...           ...
  4004e2:       89 c7                   mov    %eax,%edi
  4004e4:       e8 c4 ff ff ff          callq  4004ad <g>
```

Listing 2.13: Example of parameter passed through registers

Functions `f` and `g` (or `glib`) must use the same convention to be able to communicate: if `f` uses the first convention and `g` the second, it will lead to unwanted behaviors.

2.7.1.2 Cleaning stack

A call requires to push some values on the stack, and especially the base pointer corresponding to the state of the stack *before the call* (see Listing 2.12: `push %ebp`). The stack must be cleaned up (*i.e.*, pop these values that were pushed for the call and that become deprecated after the call sequence) when the function being called returns, otherwise the call would modify the stack frame and thus disturb the rest of the execution. Either the caller or the callee can do it, but they should not do it both. This has to be stipulate by a convention as well. These two questions in particular are solved by the specification of calling conventions. For example, the `x86 IA-32 cdecl` calling convention states that parameters are passed through the stack and the stack is cleaned up by the caller ; whereas the `x86-64 System V amd64 ABI` [MHJM13] stipulates that parameters are passed through registers. In a compiled program, every function should be using the same calling convention: at compilation time, the compiler uses either the default (OS and architecture dependent) or the specified calling convention to generate assembly code that follows the same rules. Therefore, it seems logic that multiple functions in a given binary use the same calling convention. However, for calls to library, we must ensure that both the binary and the library it uses share the same convention. In `ELF` binaries, the convention used is specified in the program header - see Listing 2.14. However, for hand-written binaries, we may want to test if this convention (specified by the header) is indeed respected. For compiled programs, tests may also be useful to verify the correctness of

compilers. For instance, in [Lin05], the author propose a method to generate test suits for several calling conventions. This method has lead, according to the paper, to the discovery of 13 new bugs in compilers relatively to function calls.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
* OS/ABI: ************************** UNIX - System V *************
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4003c0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          2624 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

Listing 2.14: Header of a `ELF` binary program (obtained with `readelf -h`

Recovering calling conventions being used in custom binaries could also be useful to apply some specific analyses techniques relying on the function parameters for instance. This (retrieving automatically the convention) seems to be a work to address, as very few papers on the subject are available. Some binaries may even use custom calling conventions (except when calling library functions, in which case they must comply with the library's convention). This could also be interesting to detect and retrieve. In summary, works on calling convention (test, retrieving, etc.) must be conducted at binary level, as this problem is invisible at source-level.

2.7.2 Stack Frames

Stack frames have also been studied and can lead to knowledge on the binary being analyzed, either on its dynamic behavior or on its design. On the stack are pushed parameters and return values (if the calling convention specifies so), intermediate values for computation, saved values of registers (especially `%ebp`), return addresses, etc. Detecting calling patterns on the stack can be helpful to retrieve information about the call stack (see Section 2.5). For instance, a

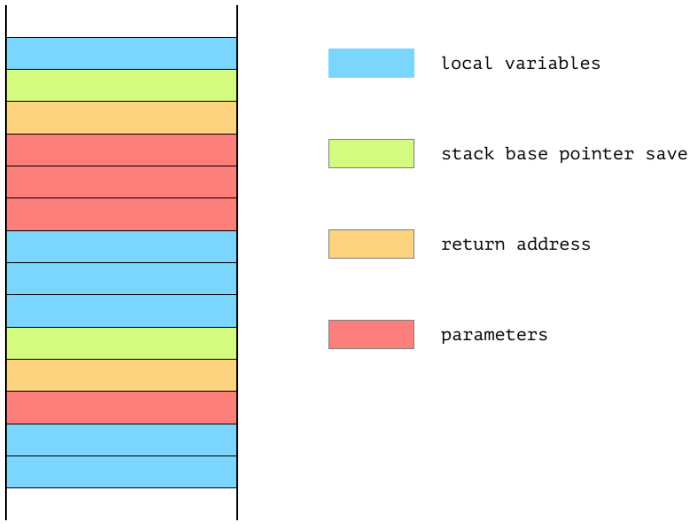


Figure 2.4: Example of stack frame patterns that allow to get an idea about the call stack

classical pattern corresponding to a call on the stack is given in Figure 2.4. By detecting this pattern in a given state of the stack, we can segment it into regions corresponding to address spaces of different functions. An application to that would be to perform verification on the stack accesses: a function should not access an address corresponding to the stack that is not in the range ⟦ `%epb` , `%esp` ⟧, except to access a stack parameter. We could also detect overriding of return addresses, which is a major issue for security (see Section 2.6). This has been addressed for instance in [KW09] and [OVB+06] at hardware level and in [CCT+15] at software level.

2.8 Miscellaneous

Section 2.2 aimed to give an overview of the many situations where working at assembly and/or binary level makes sense. This overview is not exhaustive, and

several other works focus on binaries with different purposes. Among them, we could mention profiling and code coverage.

2.8.1 Profiling

Code-profiling aims to detect the portions of code that are the most executed according to various criteria (amount of time, number of times, number of instructions, etc.). The goal is to highlight the parts that worth optimization. For instance, `GNU gprof` [Foub] is a tool to dynamically analyze the amount of time the execution spends in each function.

2.8.2 Code coverage

Code-coverage aims to detect portions of code that are actually executed, and more important the ones that are not. This can be useful in two situations:

- **test suit coverage:** used while testing a program, code-coverage highlights the parts of the code that are never executed when running the test suit. It is a convenient tool to improve the coverage of a test suit.
- **dead-code detection:** code-coverage can also be useful to detect portions of code that are never executed and therefore can be removed.

The `GNU gcov` [Foua] is one example of tools that work at binary level to output code-coverage information.

Chapter 3

Static and dynamic analysis

In the previous chapter, we have presented the main motivations and goals to analyze computer programs at binary level. This chapter focuses on *why*, presenting two main categories of analyses: static analysis and dynamic analysis. For each one, we discuss the approach, the advantages and the drawbacks, and finally several examples of research works in this domain. For a few years, techniques using a combination of both static and dynamic analyses, named *concolic* analysis - a mix of concrete and symbolic analysis, has been introduced and developed (see [SMA05] for the introduction of the concept, and [CZW14] as an example of more recent work in this direction), but we will not present this in details here.

3.1 Static analysis

Static analysis of a program is exploring possible paths of the execution without actually running it. The main aspect of this approach is **exhaustivity**. Indeed, a key element is to consider every possible value (within a specified range) for each input. Therefore, it covers every possible behavior of the program, and does not rely on coverage techniques. It also captures particular cases that are unlikely: if a given path of execution is possible, it is by the nature of the approach covered by a static analysis. However, this type of analyses cannot be performed at the scale of a program, systematically and automatically: it encounters undecidability issues. In particular, the problem of loops (how many iterations should be considered) and the termination of a program are two undecidable problems. In order to be

practical, static analyses must then use approximations to restrain the exploration to a subset where these problems become decidable. This approximation can either be an over-approximation: this allows to ensure the completeness of the results, but leads to false positives. Typically, an over-approximation can either ensure that a program is free of bugs or output some possible bugs. In the last case, it is not guaranteed that these bugs actually exist. Or the approximation can be an under-approximation: in this case, the soundness of the results is ensured, but not the completeness. It means that a bug output by the analysis will be correct, but that some bugs can be missed (false negative).

In the rest of this section, we present some basics relative to static analysis techniques, and the limitations of this kind of approach. Then we present two examples of static analysis approaches over binaries: WYSINWYX[1] [BR10], and TIE - Principled Reverse Engineering of Types in Binary Programs [LAB11], but many other works on binaries use a static approach (see [GGTZ07] and [BK12] for instance).

3.1.1 Abstract domains

As mentioned in the introduction of this section, static analysis focuses on the range of all possible values of variables, usually starting from the inputs. The range of values for input is assumed. Depending on the context and the requirements of the approach, a large set of values can be used (*e.g.*, $[\![-2^{31}, 2^{31}-1]\!]$ for a 32bit signed integer) or a more relevant subset (*e.g.*, $[\![0, 10]\!]$). The propagation of these ranges becomes a problem when two or more variables get mixed. Two main techniques have been used to answer this problem: non-relational domains, which are easy to compute but over-approximate consequently, and relational domains which present a better precision. A classical example of non-relational domains is intervals, and of relational domains is polyhedrons.

```
  // variables a (-0x8(%rbp)) and b (-0x12(%rbp)) are assumed to be in [-5;
   5]
400583:       8b 45 f8                mov    -0x8(%rbp),%eax
400586:       85 c0                   test   %eax,%eax
  // if a <= 0, jump to 400599
400588:       7e 0f                   jle    400599
40058a:       8b 45 f4                mov    -0xc(%rbp),%edi
40058f:       e8 69 ff ff ff          callq  4004fd <abso>
  // b <- abs(b)
```

[1] What You See Is Not What You eXecute

```
400594:       89 45 f4                mov    %eax,-0xc(%rbp)
  // jump to 4005a8 (after "else" bloc)
400597:       eb 0f                   jmp    4005a8 <main+0x89>
400599:       8b 45 f4                mov    -0xc(%rbp),%edi
40059e:       e8 5a ff ff ff          callq  4004fd <abso>
4005a3:       f7 d8                   neg    %eax
  // b <- -abs(b)
4005a5:       89 45 f4                mov    %eax,-0xc(%rbp)
4005a8:       8b 55 f8                mov    -0x8(%rbp),%edx
4005ab:       8b 45 f4                mov    -0xc(%rbp),%eax
  // computation of a * b
4005ae:       0f af c2                imul   %edx,%eax
```

Listing 3.1: Range analysis on a assembly instructions

3.1.1.1 Intervals

Let us consider the example given in Listing 3.1. This code takes two parameters, namely `a` (located at `%rbp - 8`) and `b` (located at `%rbp - 12`) presumably in the range $S = [\![-5, 5]\!]$, and outputs a multiplication of the two. In between, a conditional branching makes sure `a` and `b` are of the same sign (`a <= 0` $\Rightarrow$ `b` $\leftarrow$ `-abs(b)` and `a > 0` $\Rightarrow$ `b` $\leftarrow$ `abs(b)`). An analysis of ranges with intervals will not take into account the relation between `a` and `b`. In the first branch (`a >= 0`), the interval for `b` will be restricted to $S_1 = [\![0, 5]\!]$, and in the second branch (`a < 0`) to $S_2 = [\![-5, 0]\!]$. However, when the two branching merge (at address 4005a8), then the interval of possible values for `b` will be the union of S_1 and S_2, which is S. Thus, the range of possible values for the multiplication of `a` and `b` will be $S' = [\![-25, 25]\!]$, which is a significant over-approximation: in reality, the result cannot be negative.

3.1.1.2 Polyhedrons

The same example can be treated with more accuracy using polyhedrons. The idea is to keep in mind the relationship that exists between values of `a` and values of `b`. In the first branching (*i.e.*, the case where `a > 0`), `a` and `b` have values in the set $S_1^* \times S_1 = [\![1, 5]\!] \times [\![0, 5]\!]$. In the second branching (*i.e.* `a <= 0`), the set of values is $S_2 \times S_2 = [\![-5, 0]\!] \times [\![-5, 0]\!]$. When the two paths merge, the possible values for `a` and `b` are the union $S_1^* \times S_1 \bigcup S_2 \times S_2$, and thus `a*b` is in the set $[\![1*0, 5*5]\!] \bigcup [\![0*0, -5*-5]\!] = [\![0, 25]\!]$. This is still an over-approximation (primes values, *e.g.*, 17, cannot occur), but the size of the set of possible values is twice less than with an interval approach.

3.1.2 Forward and backward analyses

There are two main categories of static analysis of program: the forward and the backward approaches.

3.1.2.1 Forward analysis

The forward approach is probably the most intuitive. The analysis starts from the inputs (of a function or of the entire program), and usually map them with a set of considered values. Typically, this set is represented by an abstract domain, which could either cover all the possible values for a given type (*e.g.*, $[0; 2^{32} - 1]$ for a 32bits unsigned integer), or a more relevant subset according to some circumstantial arguments. Then, it explores (simultaneously) all the possible paths of execution, refining the set of values at each conditional branching. A simple example is given in Listing 3.2.

```
    // (1) At this point, register %ax is in [-128, 127]
    cmpb %ax, $0
    jbe $0, else
if:
    // (2) Here, %ax is in [1, 127]
    ...
else:
    // (2') Here, %ax is in [-128, 0]
    ...
```

Listing 3.2: Example of forward analysis to infer the range of possible values

3.1.2.2 Backward analysis

The backward approach is the opposite: from a given point of the program, we perform an analysis in the opposite way relatively to the normal execution, to recover paths and/or input subset of values that lead to this point.

3.1.3 Applications

3.1.3.1 Proof of programs

The exhaustivity of the static approach is often used to prove some properties on programs. Typical proofs that are targeted are the non-occurrence of some types of runtime errors, and pre-post condition guarantees.

No runtime error A static analysis can lead to the formal proof that a given type of error can never occur, or by opposition that a particular input triggers an unwanted behavior. For example, one could try to ensure that arrays are never addressed outside the bounds. Another classical application is to ensure that the second operand of a division cannot be null. An instance of the latter example is given in Listing 3.3.

```
    cmpb %bx, $0
    jbe $0, else
if:
    // (2) Here, %bx is different from 0
    // (In x86, the divisor is assumed to be in %ax)
    divb %bx
    ...
else:
    ...
```

Listing 3.3: Example of a division in assembly with non-zero divisor

A static analysis on this piece of code proves that the division between `%ax` and `%bx` cannot lead to a runtime error (division by 0).

Pre-condition and post-condition Another application of static analysis is to prove the post-condition (Q) of a function (usually), if the pre-conditions (P) are respected. For example, in Listing 3.4, consider an implementation of the absolute function.

```
00000000004004dd <abs>:
  4004dd:       55                      push   %rbp
  4004de:       48 89 e5                mov    %rsp,%rbp
  4004e1:       89 7d ec                mov    %edi,-0x14(%rbp)
  4004e4:       83 7d ec 00             cmpl   $0x0,-0x14(%rbp)
  4004e8:       79 0a                   jns    4004f4 <abs+0x17>
  4004ea:       8b 45 ec                mov    -0x14(%rbp),%eax
  4004ed:       f7 d8                   neg    %eax
  4004ef:       89 45 fc                mov    %eax,-0x4(%rbp)
  4004f2:       eb 06                   jmp    4004fa <abs+0x1d>
  4004f4:       8b 45 ec                mov    -0x14(%rbp),%eax
  4004f7:       89 45 fc                mov    %eax,-0x4(%rbp)
  4004fa:       8b 45 fc                mov    -0x4(%rbp),%eax
  4004fd:       5d                      pop    %rbp
  4004fe:       c3                      retq
```

Listing 3.4: Implementation of the absolute value function

The input parameter, given through register `%edi`, is first stored on the stack (`mov %edi,-0x14(%rbp)`). Then, it is compared to the numeric constant `0`, and two cases are to be differentiated:

- if it is lesser than or equal to zero: the instruction `jns 4004f4 <abs+0x17>` will not be executed. Then, the value corresponding to `-%edi` is returned;
- if it is strictly greater than zero: the instruction `jns 4004f4 <abs+0x17>` will be executed, and the value `%edi` is returned.

These two cases are exhaustive, and so we can conclude that this function always returns `abs(x)` where `x` is the given parameter.

3.1.3.2 Exploration

Static analysis is also used to explore the possible behaviors of the program, and in particular the possible values a variable can have at a given point of the program ; or the possible paths that can be activated, and with which concrete values in input.

Value range This has been partially covered in a previous section. The value-set analysis aims to provide information about the possible values of a given variable at a given point of a program. Typically, it will be used to get the range of possible values for the output of a function, regarding a range for inputs. This sort of analysis can be done either with intervals or polyhedrons, by propagating the range from inputs through the execution of the function until every possible path reaches a return statement.

Path activation The path activation problem can be summarized with the following question: what value for each input leads to reach this particular state of the program?, where a state is often the entry of a basic block. From an attacker point of view for example, an instance of this problem would be to get the proper input (password) to reach the basic block "authenticated".

3.1.4 Limitations

The main limitations of a static approach are the following:

- The problem it aims to solve is undecidable ; therefore the practical use of this technique requires approximations that lead either to false positive (loss of soundness) or false negative (loss of completeness).
- Because it aims to be exhaustive, multiple branching can lead to an explosion of the number of paths to cover, and so be a limitation of the exploration in practice.
- Static analysis is vulnerable to obfuscation. In particular, if the core of the program is packed (for instance using `UPX` [OMR04]), it cannot be analyzed statically without complications. Another typical obfuscation that limit static analysis is opaque predicates (see [CTL98] for instance).

3.1.5 WYSINWYX [BR10]

In [BR10], the authors propose a static analysis of binaries to ensure security properties on the program being analyzed.

3.1.5.1 Binary code rather than source code

They emphasis the importance of working on binaries, as the compiler, through optimizations, can induce behavioral differences between what is intended at source-level and what is actually executed by the CPU ("What You See Is Not What You eXecute"). The given example is the following:

```
memset(password, '\0', len);
free(password);
```

This code is used in Microsoft Windows ([How02]) to override sensitive data (here, the password stored in clear in memory) before freeing the memory. A compiler, for optimization purposes, could consider this call to `memset` useless, as the memory location being written is never read after this point, and thus remove this call.

3.1.5.2 Value-set Analysis

The first step of the proposed approach is to track both numeric and address values, using an over-approximation of the set of possible values and constant

values declared statically. Then, using the address values are used to reconstruct a call graph and a control-flow graph (with the limitations due to the indirect calls that are hard to evaluate statically). It also reconstruct an abstract-syntax tree and a system dependence graph.

3.1.5.3 Model-checking

From the results of the Value-set analysis, they propose a *weighted pushdown system* to model possible program behaviors, with an abstraction of the runtime states. From this, the authors address the problem of reachability of control states, and especially undesired ones. They also use an automated generator of queries to provide witnesses that activate execution path driving to an error configuration.

3.1.5.4 Limitations

As mentioned by the authors, this work suffers from the limitations of static approaches. In particular:

- the approach is not really scalable, and is not conveniently applicable on large binaries,
- the lake of concrete values lead to approximations, especially relatively to the call graph and the control-flow graph.

3.1.6 `TIE`: Principled Reverse Engineering of Types in Binary Programs [LAB11]

[LAB11] presents a "type reconstruction system based on upon binary code analysis". The approach can be performed either statically or dynamically, only the first step differ. However, we focus here on the static approach.

3.1.6.1 Intermediate Representation

In this approach, first the authors translate the binary code into an intermediate representation, in this case `BIL` (Binary Intermediate Language). In the case of a dynamic analysis, they translate the sequence of instructions that were executed during a particular trace.

3.1.6.2 Variable recovery

From the intermediate language instructions, authors retrieve variable locations using a SSA/DVSA algorithm. The SSA (Static Single Assignment) algorithm aims to denote with incremented names the memory locations that store successive values. For instance, a first assignment of the register `%eax` (*e.g.*, `mov $0, %eax`) will be rewritten into an assignment of register $\texttt{\%eax}_0$ (`mov $0,` $\texttt{\%eax}_0$) ; and a second assignment of the same register later in the execution (*e.g.*, `mov $0xcafe, %eax`) will translate into an assignment of $\texttt{\%eax}_1$. This allows to deconflict multiple variables handled by the same register. Then, they use a custom Value-Set Analysis (VSA) algorithm to output variable locations plus alias information, and sets of possible values.

3.1.6.3 Variable typing

From the location of variables, and considering the operations performed on them, the authors deduce an upper and a lower bound of the type of each. To do so, they propose a hierarchy of types where `a < b` if `b` is less restrictive than `b` (for instance, `num32_t < int32_t` and `int32_t < int16_t`). The hierarchy they use has a depth of 5, including the "any type" level (the lower one) and the "inconsistent type" level (the higher one). The latter one corresponds to an over-constrained type (`e.g.` signed and unsigned in the same time). This can occur for instance when `union` are used in `C` programs. As an example, a signed division leads to the restriction to signed types for each of its operands.

3.1.6.4 Evaluation

Regarding accuracy, the approach shows good results: in over 90% of the cases, the inferred type bounds include the source-level type of the variable, with an average length of interval of 2. However, the authors do not provide any information about the cost (in time and in memory) of their approach.

3.2 Dynamic analysis

In opposition to static analysis, a dynamic analysis focuses on a restricted number of behaviors of the program under analysis: the ones that can actually be observed during an execution. The main advantage of this technique over static analysis is

that it excludes by design the possibility of false positive (although exceptions can be forced - see Section 3.2.2). Indeed, as it works on concrete executions, what is observed can necessarily occur. The main downside effect is that it faces the problem of coverage: where static analysis aims to be exhaustive, dynamic analysis is limited by the traces it deals with. Because dynamic analysis works directly on the execution of a program, it opens the possibility to prevent an undesirable state to be reached with limited effort (for instance by stopping the execution when the undesirable state is about to be reached). The same kind of enforcement with static analysis will require to explore the possible paths of execution, and to filter the inputs in order to make this state unreachable by implementation, which can be much more expansive. The cost of dynamic instrumentation can be chosen: it is possible to instrument only small parts (*e.g.* a given function) of a program instead of every instruction. Some parts of the program (for instance library calls) can be considered as black-box elements ; which is not possible to do with a static approach. Finally, a dynamic approach works with the concrete environment of the program. In particular, system calls, available memory, etc. are not to be assumed or modeled as they should be in static analysis. The downside is that the results obtained are highly context-dependent, whereas static analysis provides more general results.

In the next sections, we present some specificities of the dynamic approach, and then we discuss into more details the limitations of it. Finally, we present two major works in this domain: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [NS05] and Howard: A Dynamic Excavator for Reverse Engineering Data Structures [SSB11].

3.2.1 Instrumentation

Instrumentation techniques are the keystone of a dynamic analysis. The way instrumentation is handled defines the possibilities given to the analysis (both in terms of inspection and in term of control), and impact directly the overhead at runtime. Nowadays, the main dynamic instrumentation frameworks use *just-in-time* (JIT) compilation to provide high capabilities with a reasonable overhead. This is the case for `Pin` [LCM+05], `Valgrind` [NS03] and `DynamoRIO` [BDB00] in particular. JIT consists in re-writing, at runtime, parts of the binary that are about to be executed, enhanced with instrumentation instructions. The main advantage of this technique, despite the fact that "compilation" occurs in the name, is that it does not require recompilation of the program to analyze, and thus no source

code is needed. It also allows to instrument dynamically loaded libraries.

Another class of tools to perform dynamic analysis are debuggers. On Linux systems, the most famous is `gdb` [GSS90]. A debugger's approach is very different from a framework such as `Pin` using JIT: it usually attaches itself to the program to analyze (*e.g.*, using `ptrace`), allows the user to put some software breakpoints (replacing an instruction with `INT 3`) or hardware breakpoints (provided [or not] by the CPU), and then start the execution. The debugger will be able to get the focus back only when the program stops or a breakpoint is reached. At this time, it is possible to get concrete values stored in registers, explore the stack and possibly alter these values. The use of breakpoints, however, makes the automation of dynamic analysis hard to perform: it requires to anticipate the execution, enough to be able to stop the program when needed. In addition, a breakpoint leads to a context switch (from the program under analysis to the debugger), which is costly in time.

3.2.2 Observation vs. Modification

The instrumentation of an execution can either be passive or active. Choosing either one or the other has consequences on the soundness, the coverage, etc. of the results.

3.2.2.1 Passive instrumentation: observation

Passive instrumentation consists in observing, at runtime, what is observable, without interfering with the execution. Typical observations focus on (not exclusive):

- **register values**: return value of functions (`%eax`), program counter (`%pc`), stack frame (`%ebp`, `%esp`), etc.
- **stack inspection**: local variables, input parameters (according to some calling conventions), return address, etc.
- **heap inspection**: the number of allocated blocks, their content, etc.

In most cases, a passive instrumentation does not modify the behavior of the program under analysis. This is the main advantage of this approach. However, in some cases, anti-debugging and anti-instrumentation techniques can lead a program to change its control flow if it is being instrumented (see [CAM+08] and [BD06] for instance). This is a limitation of the dynamic approach that will

be discussed into more details in Section 3.2.4. The drawback of this passive instrumentation is that the observation is confined where the normal execution actually goes, and suffers from the problem of coverage (also discussed in Section 3.2.4).

3.2.2.2 Active instrumentation: modification

Another approach is to perform instrumentation that voluntarily influence the execution. For example, one can force the program to execute a conditional branching even if the conditions are not met. The main interest of this is to easily explore paths without finding inputs that actually activate them. This reduces significantly the problem of coverage relative to dynamic analysis (although it does not eliminate it completely), but also introduces the risk of false positives, that are in theory not an issue with a dynamic approach. Indeed, by forcing some conditions despite the concrete values at runtime, one could drive the program to a state that is in fact impossible to reach without active instrumentation (*i.e.*, only with legitimate inputs).

3.2.3 Strategies

3.2.3.1 Runtime enforcement

As mentioned in the beginning of Section 3.2, dynamic analysis allows to perform some runtime verification, and even prevent some misbehaviors of a given program to actually occur. For instance, in [Cow98], the canary-based protection against buffer overflows detects at runtime an overriding of the return value, and prevent the program to continue its execution. In [Ber06], the authors do similar prevention regarding heap overflow. In [You15], use-after-free vulnerabilities are also detected at runtime, by dynamically perform additional checks before accessing a memory location.

3.2.3.2 Online and Offline analyses

Another approach of dynamic analysis consists in two steps. The first step, called *online step*, consists in collecting data (flow of execution, values of registers, call stack, etc.) during the execution of the program. During a second step, called *offline step*, this collected data is analyzed and from that some deductions are made about the execution.

This two-steps approach usually allows to execute the program under analysis with a reasonable overhead, however it cannot be used to prevent misbehaviors of programs for instance (*e.g.* crashes, vulnerability activation, etc.).

3.2.4 Limitations

As presented in the introduction of this Section 3.2, dynamic analysis by design presents some limitations.

- The main limitation of a dynamic approach is **coverage**. Every path that is not covered by the instrumented executions cannot be analyzed. As mentioned in a previous paragraph, this limitation can be slightly reduced by performing active instrumentation, and in particular by forcing conditional branching. However, doing this may lead to explore paths that are impossible to go all over in practice.

- Results of dynamic analysis are dependent on the context of the execution. This leads to two sub-limitations. First, it is hard to generalize the results that are obtained, because they may depend on some context-dependent parameters that one is not aware of (for instance, the state of some uninitialized memory). Second, and for the same reasons, replay may not always be possible. For instance, a program reading a non-initialized variable may or may not crash, depending on the value of the uninitialized memory location.

- Dynamic analysis requires to be able to actually execute the program under analysis. This means 1) that we can provide a context in which the program can indeed be executed (*i.e.* the relevant architecture, needed libraries, etc.), and 2) that either the program is trusted or we can confine its execution to a restricted area in which it cannot make damage.

- Because the program under analysis is executed, it can detect and try to defend actively against analysis. For instance, it can hide a part of its behavior if it detects that it is being analyzed. This anti-debugging and anti-instrumentation techniques are used by many malwares, and even by legitimate programs such as [BD06].

3.2.5 Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [NS05]

In [NS05] (2005), the authors propose to adapt the taint analysis technique to a dynamic approach to detect vulnerabilities on binaries. Their approach was a significant improvement of the state of the art, as it is applicable to binaries with no source needed (neither specific compilation options) and presented qualitative results. This work has been reused a lot since ([CW08], [ARF+14] for example). Their main objective is to detect at runtime when an untrusted input (*e.g.* network packets, user input, etc.) are used in a dangerous way. They propose three steps: `TaintSeed`, `TaintTracker` and `TaintAssert`. The taint mechanism relies on a shadow memory.

3.2.5.1 Shadow Memory

A shadow memory maps every byte of the virtual memory used by a program to one or several bytes of an "abstract" memory that only stores meta-data. In the case of [NS05], they map every single byte of the program under analysis to four bytes in the shadow memory. These four bytes are used to store a pointer to a data structure which stores information about the source of the content of the data (trusted or not, provenance, etc. - see next paragraph).

3.2.5.2 TaintSeed

The first step is to taint untrusted data. Every data coming from network sockets, files or `stdin` are marked as untrusted, and a `Taint` data structure is allocated in the shadow memory to record meta-data (snapshot of the current stack, copy of the concrete value of the data, etc.).

3.2.5.3 TaintTracker

The second step is to propagate the tainting from the untrusted sources when they are used in computation or memory accesses. They differentiate two types of instruction:

- *arithmetic instructions* (*e.g.* `ADD`, `XOR`, etc.): the result of an arithmetic instruction will be tainted if and only if any byte of any of its operands is

tainted.

- *data movement instructions* (*e.g.* `LOAD`, `STORE`, etc.): the data at the destination will be tainted if and only if any byte of the source is tainted.

3.2.5.4 TaintAssert

Finally, `TaintAssert` instruments every operation that could lead to an illegitimate behavior. In particular, for each instruction that modify the control flow, it checks if the instruction's operand is tainted. For instance, it checks that the data specifying a `JMP` target is tainted. If so, then an anomaly is detected and the execution stops. An `ExploitAnalyzer` step outputs details about the execution that led to this anomaly (for instance, the provenance of the tainted data that activated the detection).

3.2.5.5 Evaluation

The authors monitored the execution of several programs, as `apache`, `ATPhttpd`, `ssh`, `gcc`, `vim`, etc. They claim that no false positive was detected during the execution, and that functionalities of these programs where not victims of the instrumentation. On `ATPhttpd`, they were able to detect a buffer-overflow attack that targets the return value, and on synthetic programs they were able to detect buffer overflows on function pointers and format strings. Regarding performances, the announced overhead is between `x1.5` and `x40`.

3.2.6 Howard: A Dynamic Excavator for Reverse Engineering Data Structures [SSB11]

In [SSB11], Asia Slowinska et al. present a dynamic approach to retrieve data structures from the execution of binaries. In particular, they target variables, arrays and structures, and aim to use behavioral patterns to differentiate the three cases. The problem can be illustrated by one simple example: let us consider an instruction that accesses `*(A + 4)`, *i.e.* where `A` is a memory location. `(A + 4)` can either point to:

- **a variable** - *e.g.* `(%ebp + 4)` is typically the location of a local variable ;
- **a cell** - *e.g.* if `A` points to an array of integers ;

- **a structure field** - *e.g.* the second filed of a structure whose first field is an integer.

The first part of the problem addressed in this work is to distinguish these three cases.

3.2.6.1 Function call stack

First, they instrument `CALL` and `RET` functions to keep a consistent call stack. This is important to contextualize accesses, in particular to the stack (an access to `*(%ebp + 4)` highly depend on the value of `%ebp`, which changes at each call). As we discuss in Chapter II, the problem of keeping a call stack consistent is not trivial, and deserves to be addressed.

3.2.6.2 Pointer tracking

The next step is to keep track of pointers used to access data. The main idea is to store, for each memory location `B` used at runtime, how it was computed: if it is derived from another pointer, then the tag `MBase(B)` is set to `A` ; if it was derived from no other address, then `B` is considered as a root pointer, and then `MBase(B)` is set to *root*.

3.2.6.3 Multiple base pointers

A given memory location `B` may be addressed using different base pointers. For instance, `memset`, often used to initialize dynamically allocated blocks, accesses the data with no understanding of the structure. It will typically iterates, from the base pointer `B`, by incrementing a pointer several times (and then each memory access is computed using the previous address as a base pointer). Later accesses will use the base pointer to access fields, according to the structure definition. To deal with these cases, the authors propose heuristics to keep the less regular structure (*i.e.*, for instance, structures with fields of different sizes over arrays).

3.2.6.4 Detecting arrays

To detect arrays, they distinguish two main scenarios.

- **Access relative to previous element** - typically a loop that increments a pointer at each iteration.

- **Access from the base pointer** - typically an offset incremented at each iteration.

3.2.6.5 Evaluation

Howard presents encouraging results, as it recovers over 90% of the structures and arrays in `wget` and `lighttpd` for instance.

Chapter 4

Problems addressed

In this section, we present our own choices for the approach we propose in Part II. In Section 4.1, we present our objectives, and in Section 4.2 the main choices we made regarding the approach to consider.

4.1 Objectives

4.1.1 Short-term: find points of interest in a program

Manual analysis of a binary is fastidious, mainly because it requires efforts to understand machine code for a human being. We aim to provide an approach that would help a reverse-engineer focus rapidly on the main points of interest of the program. In this way, we aim to provide **structure** information to get a quick understanding of the program structure ; and **behavior-level** information (*e.g.*, information about the control flow and the data flow, the way memory is managed, etc.) to emphasis interesting points (*e.g.*, points where a suspicious behavior is detected).

4.1.2 Long-term: automatically detect vulnerabilities

A long-term objective is to provide a fully automated vulnerability detection framework. This is not in the scope of this work, but the remaining steps to climb seem accessible. An interesting work, for example, would be to merge our approach with

approaches that specifically focus on vulnerability detection but which require more information that the ones that are accessible from a binary at first look.

4.1.3 Criteria

Our objectives, explained in a general way in the two previous sections, can be illustrated with three main criteria, that we detail in the following sections:

1. **accuracy** - *we want to present accurate results*
2. **universality** - *we aim to target as many binaries as possible*
3. **scalability** - *the approach needs to be usable in practice, even on large programs*

4.1.4 Accuracy

In this approach, we aim to produce accurate results in practice. This criterion will be validated through experiments. In addition, we try to limit false positives: when possible, we favor under-approximations rather than over-approximations.

4.1.5 Universality

The approach we propose aims to be applicable to the largest set of programs. This leads to several choices. First, we target **binaries**, and do not rely on the knowledge of any other information, from the source code or from anywhere else. The input of our approach is a binary, and only a binary. This allows to deal with proprietary or malicious applications, for which the source code is not available, and not only on open-source programs. Second, we do not rely on any particular compilation option, nor on debugging information that may be embedded in a binary, although we need to know the calling convention. Third, we do not rely on the symbol table, thus this approach can deal with stripped binaries as well.

Discussion This universality is, in fact, relative to a given category of programs (that one can call the class of kindly programs). On the other hand, programs can be designed to be hard to analyze (on purpose), through obfuscation, self-modifying code, packing, etc.

4.1.6 Scalability

We want to propose a scalable approach. In order to be practical, and as this is a first step in a larger analysis process (either manually - Section 4.1.1, or automatic - Section 4.1.2), it is important to provide data in a reasonable amount of time, even on large programs such as PDF viewers or text editors.

Discussion The scalability we target is at the scale of a common program. However, we do not aim to be able to perform analysis at the scale of an operating system for instance. By "reasonable amount of time", we mean that the execution can still be performed within a few seconds (and not more than minutes)[1]..

4.2 Design choices

4.2.1 Dynamic analysis

Among the two main categories of analysis that can be performed on binaries (see Sections 3.1 and 3.2), we choose to favor the dynamic approach, with its pros and cons as mentioned in Section 3.2.4. In particular, we believe this approach is more **scalable** than a static one. Regarding the objective of **universality**, this has two consequences.

- + We can analyze programs that are statically obfuscated (up to a certain point) with small effort. For instance, packed binaries and opaque predicates are less limitative than with a static approach.
- – We cannot analyze programs that we *cannot* (because of compatibility issues, lack of libraries, etc.) or *do not want to* (because we suspect it to be malicious) execute.

4.2.2 Function-grained approach

A classical way to perform dynamic analysis is to work at the scale of instructions. In this work, we propose to focus on a higher-level element of structuration: functions. Although we do perform instrumentation on instructions, every analysis we propose

[1] *i.e.*, one should not be afraid of replaying an instrumented execution to test a new input for instance

will be based on functions. For instance, we define a data-flow at function-level, and we propose to analyze the memory management of a program by observing function calls and returns only.

4.2.3 Passive instrumentation

Another choice we make is to keep the instrumentation passive. Indeed, we do not want, as much as possible, to interfere with the execution of the program under analysis, in order to catch a behavior that is as close as possible to its behavior when it executes without analysis. This is justified by the fact we attempt to limit false positives. As mentioned in Section 3.2.3, interfering with the execution can lead to an over-approximation, because we may explore paths that cannot exist in practice.

4.2.4 Online and Offline steps

The **scalability** criterion induces an effort to minimize the total time of analysis of a program. However, the time of execution overhead and the memory overhead are the main criteria for scalability. Therefore, these are the first measures we aim to reduce as much as possible. We propose to adopt a two-steps approach: a first online step which consists in collecting data, and a second offline step to deduce information from these data. The online step has to be as lightweight as possible, to keep a reasonable overhead. Of course, the offline step must also be efficient in time, but this is not as critical. Indeed, the offline step can be replayed (for instance with different parameters or heuristics) and can be subject to optimizations such as parallelism.

4.2.5 A single execution

A major particularity of our approach, which also pursue the objective of **scalability**, is that we rely on a single execution for each analysis. Many dynamic approaches use multiple traces, whereas we aim to work with one. We claim that it is possible to recover, with high accuracy and a low overhead, a lot of information in a single execution. This highly exposes our approach to the coverage problem (as in one execution, it is very likely that all branching have not been taken), but the main advantage is that we keep it lightweight. In a future work, we could include fuzzing techniques to cover more path of execution and therefore improve the coverage of

our results, but we should emphasis the fact that multiple executions of the same control flow (even with concrete values that differ) are not required to retrieve accurate information. Coupled with a low overhead of the instrumentation, the single execution makes the proposed approach indeed scalable by design.

Part II

Dynamic analysis of binaries

Chapter 5

Definitions

In this chapter, we introduce definitions and notations that we will use in the rest of this work to describe the problems we address. We aim to define at binary level notions that only exist explicitly at source level, such as functions. Our challenge is to propose definitions that are general enough to cover the largest set of binaries we can encounter, independently from an architecture or the way it was produced (compilation, hand-written), but that are still consistent with their equivalents at source-level in the case of compiled programs.

5.1 Generalities

In this section, we present some general notations and notions that we will use in the following chapters. We propose our own definitions of execution and binary in order to be as much universal as possible: they seem to be applicable to a large variety of cases, with no assumption on the form a program takes. Our approach is based on the dynamic behavior: we consider a program through the executions we observe. From these executions, sometimes we can retrieve a static form we call binary, but not always. Our definitions allow to consider, for example, a program that would never be stored in its whole in any local memory (*e.g.*, a program that gets a part of the instructions to be executed from a remote host).

5.1.1 Instruction

We call *opcode* a basic operation in a considered machine language (*e.g.*, `ADD`, `LOAD`, etc.). An *opcode* can have zero, one or several *operands*. We denote by $|opcode|$ the number of operands an opcode takes. It can be zero (*e.g.*, $|\texttt{RET}| = 0$), one (*e.g.*, $|\texttt{JMP}| = 1$) or several (*e.g.* $|\texttt{ADD}| = 2$) operands. From opcodes and operands, we define the notion of instruction.

Definition 1. *An instruction is a couple (opcode, operands), where operands is a list of $|opcode|$ operands.*

For example, $(\texttt{ADD}, [\$2, \%eax])$ is an instruction corresponding to the addition of the constant value \$2 with the value of the register $\%eax$. We denote by *size(ins)* the size of an instruction *ins* (*i.e.*, the size of the opcode plus its operands) computed statically.

5.1.2 Execution

We define a sequence of instructions as several instructions dynamically executed, and indexed by an address.

Definition 2. *A sequence of instructions is a list of couples (addr, ins), where ins is an executed instruction and addr is the address where ins is loaded in memory. Addresses are totally ordered and pairwise disjoint.*

Instructions are loaded from a base address we denote by *base*. This base address is dynamic. From *base*, we can compute the offset of each instruction in the sequence: *offset = addr - base*. Finally, we define an execution as a sequence of concrete instructions plus a base address.

Definition 3. *We call execution a tuple (base, sci), where base is the base address where instructions are loaded and sci is a sequence of concrete instructions, i.e., instructions plus the (dynamic) concrete value of their operands at the time they are executed. The address of the first instruction of sci is called entry point of the execution.*

For example, $(\texttt{ADD}, [\$2, \%\texttt{eax}])$ is an instruction and $(\texttt{ADD}, [\$2, \%\texttt{eax}], [\$2, \$1])$ is a concrete instruction (where $\%eax$ has the value \$1).

By abuse of notation, we say that an instruction *(addr, ins)* is in an execution e ($(addr, ins) \in e$) if there is a concrete instruction in e that corresponds to

(addr, ins), *i.e.*, the address and the static instructions are the same, and without considering the concrete values of the operands.

5.1.3 Binary

From several executions $e_1, e_2, ...e_n$ of a given program, we try to get a consistent static description we call *binary*. The intuition here is that if for every observed execution, a given offset always corresponds to the same instruction, then we can construct a static representation of all these executions where an offset corresponds to an instruction. Note that this static representation may depend on the number and the nature of the considered executions.

Definition 4. *If, for n executions* $e_i = (b_i, s_i), i \in [\![1, n]\!]$ *of a program with the same entry point ep, we have*

$$\forall offset, \forall e_i = (b_i, s_i), \forall e_j = (b_j, s_j),$$
$$[(b_i + offset, ins_i, vals_i) \in s_i \wedge (b_j + offset, ins_j, vals_j) \in s_j] \implies [ins_i = ins_j]$$

then we denote by B and call binary the set of couples $(offset, ins)$*:*

$$B = \bigcup_{e_i} \{(addr - b_i, ins), \forall (addr, ins, vals) \in s_i\}$$

We denote by $\mathcal{E}(B)$ *the set of possible executions of B starting at entry point ep.*

In other words, we check that for every address $addr_i$, there is at most one corresponding instruction, if we consider every execution $e_i \in e_1, e_2, ...e_n$; and if so, we construct a static representation by the union of every tuple $(addr, ins)$ seen during one of these executions. From this point, we only consider programs for which a static representation can be computed, and unless if explicitly stated, we make no further distinction between a program and its binary representation.

Remarks A self-modifying code, for instance, will not correspond to our definition, as we can have different instructions executed corresponding to the same offset. In these cases, it seems logic not to consider one static representation of the binary. We can also imagine a program that reads its instructions through a network socket.

5.1.4 Memory location

During an execution, data can be accessed, either in memory through addressing, or in registers. We define the set of memory locations, *i.e.* every location accessible by the program that can hold data during an execution e.

Definition 5. *The set of memory locations relatively to a given execution e of a binary B, denoted by $\mathcal{M}^*(e)$, is the set of locations where the program B can read and/or write.*

This definition includes the whole addressable memory, plus registers.

We also need to define the content of a memory location. When the size of the data is known, then the content of a memory location is explicit. However, it can be hard to know the size of the data, and in this case we still need to express properties of the memory location content. To do so, we define a minimum size.

Definition 6. *The minimum size of a memory location is one byte.*

From this, the minimum content of a memory location corresponds to its content if the size is known, and to the first byte if not.

Discussion For registers, the "first" byte is the eight least-significant bits, and for memory locations pointed by an address a, it corresponds to the byte pointed directly by a. Note that there can be more than one byte of actual data being passed, but if we ignore the exact size we assume that it is at least one byte.

5.1.5 Program counter

We denote by $\%\mathrm{pc}$ the *program counter*, *i.e.* a memory location that contains the address of the next instruction to be executed. The value of this *program counter* changes after each instruction executed, and there are two scenarios to distinguish:

- either it is incremented by "one instruction": $\%\mathrm{pc} = \%\mathrm{pc} + \textit{size(ins)}$, meaning that the next instruction to be executed will be the next instruction in order of addresses - we call this scenario **sequential execution**,
- or its value is modified by the instruction being executed, either relatively to its current value or in absolute, but in a way that is not equivalent to the sequential execution - we call this scenario **discontinuous execution**.

Remark - If, for instance, the %pc value is set by a JMP instruction that happens to point to the next instruction sequentially, we still consider the execution as sequential.

5.2 Structure

We define here elements relative to the structure of a binary. The definitions we give are general, and can be applied to any binary with no assumption on its provenance. We show, for each of them, that it captures in particular structures that make sense in a binary obtained by compilation, regarding the structures that can exist at source-level, but our approach is not limited to compiled binaries. Other definitions of functions, parameters and type exist, but are always related to source-level information. For instance, in [ASB17], they define entry and return points in a binary relatively to the source code. Here, we propose definitions that are applicable to binaries that were not obtained by compilation. They are also applicable to executions that do not correspond to a static binary file.

5.2.1 Function

There is no proper notion of functions at binary level, although they are key elements of the structuration of programs at source level. We propose to define functions at binary level, and to use it as an element of structuration as well, which would be consistent with functions at source level in the case of compiled programs. This consistency is specifically discussed in Section 5.2.1.4.

5.2.1.1 Definitions

As we use a dynamic approach, our definition of function is relative to a given execution.

Call First, we need to define a *call*.

Definition 7. *A call c during an execution e is either:*

- *a block of instructions executed sequentially such that: the first instruction saves the address of the instruction following the block, denoted by ret(c), and the last instruction induces a discontinuity in the evolution of* %pc*, from its current value to a new address we call the target of the call;*

- *or an assignment of* %pc *to the target of a previous call.*

Stack of calls We represent calls by the construction of a stack of calls, that we denote by $S(e)$ for a given execution e. Each time a call is detected during e, we push $ret(c)$ in $S(e)$. The following events can lead to pop values from $S(e)$:

1. if the execution e terminates, then every value of $S(e)$ is popped,
2. if %pc takes a value v corresponding to a $ret(c) \in S(e)$, then values are popped from the top of $S(e)$ until the first occurrence of this value is found (this value is also popped),
3. if a given $ret(c)$ is overwritten, then values are popped from $S(e)$ until $ret(c)$ is reached (this value is also popped).

Event 1 corresponds to a (possibly unexpected) termination of e at a point where $S(e)$ is not empty. Event 2 means that when a *call* c restores the value of %pc, *i.e.* the sequential execution (before the call) resumes, every call that occurred after c is also popped from $S(e)$. Finally, event 3 means that when there is no more way a given call c can restore %pc to the value it had before c, then $ret(c)$ and $ret(c')$ for every c' that occurred after c are popped from $S(e)$.

Return From these definitions of *call*, we can define a *return* corresponding to each call.

Definition 8. *A* return *during the execution e, corresponding to a call c, is when $ret(c)$ is popped from $S(e)$.*

Now that we defined a *call* and its corresponding *return*, we can define a *function* in a binary, relatively to an execution e.

Definition 9. *We call* function *a target to at least one call during e. A function is identified by the index (address) of its first instruction, i.e. the target of the call.*

We denote by $\mathcal{F}(e)$ the set of functions, according to Definition 9, and relatively to the execution e of the binary under analysis. Finally, for every function $f \in \mathcal{F}(e)$, we define executions of f.

Definition 10. *We define an execution of f, denoted by ε, the sequence of instructions executed from a call of f to the corresponding return.*

Note that this definition includes instructions corresponding to another function that would be called during the execution of f.

Definition 11. *For a given execution e, and a function $f \in \mathcal{F}(e)$, we denote by $e\lfloor f$ the set of executions ε of f that occur during e. By extension, we denote by $\mathcal{F}(\mathcal{E}(B))$ the set of executions of f occurring in any execution e of the binary B.*

5.2.1.2 Discussion

Call The two-steps call we define in Definition 7 allows to distinguish a simple transfer of the control flow (*e.g.* a jump in a loop) from a more complex structure we call *function*. The particularity of functions we choose to emphasis is the capability to return, after the transfer of the control flow, to where the execution was. This capability requires a save of the program counter (or the next sequential instruction) before transferring the control flow, and that is what characterizes *calls*. Without this notion, functions and basic blocks would be indistinguishable.

Return Our definition of return raises several points that deserve to be discussed. First, we consider that every function being called returns at the end of the execution e of the program under analysis. Second, we consider that if f calls g and g calls h, then when f returns, both g and h return. Finally, we consider that a function returns when the saved value of %pc is not available anymore. It can be because it has been consumed to return to the point of the execution where the call occurred, or because it has been lost. In any case, the function cannot return anymore to the initial point of the execution, and therefore we consider that its own execution is over.

Function identifier According to Definition 9, we identify functions with the index of their first instructions. This has several consequences. First, it means that a function has a unique entry point (which is its identifier), and that this definition excludes to consider functions with multiple entry points as the same structure. Second, as the sequence of instructions being executed from the call to its corresponding return is not a part of the function identifier, this definition covers functions that are made of multiple basic blocks and conditional branching: several sets of executed instructions starting from the same index will be identified as the same function. Finally, our definition of return corresponding to a call does not imply uniqueness. Therefore, it deals with functions that have multiple return

points, This means, in particular, that the functions we capture can be made of multiple return points in different basic blocks.

5.2.1.3 Examples

We propose several examples and counterexamples of calls and returns to illustrate our definitions. We use, as much as possible, x86 to give code examples, but in some cases we take liberties, and especially we assume that the program counter can be modified by classic instructions (`mov`, `add`, etc.) whereas this is not possible in x86.

Call According to our definition, a call is a save of a value of %pc to be able to return and a discontinuity in the execution. We give here two examples which do that at assembly level.

Instruction `CALL` The more intuitive example of call, in x86 for example, is the instruction `CALL`. Indeed, this instruction performs a push of the program counter on the stack, and a jump to the targeted address.

```
call 80484cd <f>
```

Explicit save of the program counter Assuming that it is possible to manipulate directly the program counter %pc in a given architecture, a sequence of instructions starting by pushing %pc + ... (depending on the number of instructions) and ending with an absolute jump would also correspond to a call according to Definition 7:

```
mov %pc, %eax
add 3, %eax
push %eax
jmp @...        // target of the call
```

Note that this save of %pc could be done through registers instead of the stack (but in this case, there can only be one depth of call).

Return We recall that a return occur when either the program counter takes back the value it had before the call, or when this value is not accessible anymore (if we exclude the case where the execution stops before functions have returned).

Instruction `RET` The instruction `RET` in `x86` is also a good candidate for returns, as it pops the address where to return from the stack and set the program counter to this value. It is generally preceded by a restoration of the stack base pointer (`%ebp`), but this is not a requirement.

```
pop %ebp
ret
```

Explicit return Once again, assuming that the program counter can be read and written through classic instructions, a return can be implemented by several other ways. For instance, a simple move from the location of the save to `%pc` implements a return. Another one would be an override of this location.

```
pop %eax       // we assume that the return addr
jmp %eax       // was saved on the stack
```

5.2.1.4 Consistency with `C` compiled binaries

This definition of functions is compatible with `C` compiled binaries, and is consistent with the notion of function at source level in `C`. We discuss in this section some particular points.

`CALL` **and** `RET` The most frequent way a function call is compiled by `gcc` (and other compilers) from `C` to assembly is using the instruction `CALL`. In `x86`, for instance, this instruction pushes the current value of `%pc` (plus an offset to return to the instruction that follows `CALL` and not `CALL` itself), and changes the value of `%pc` (with the address of the targeted function). It corresponds exactly to the two-steps event we defined. In the same way, the `RET` instruction pops the return address from the stack (and therefore consumes it) into `%pc`.

Compiler optimizations In addition, our definition also includes compiler optimizations in the way it compiles calls and returns. For instance, some calls are performed using a `JMP` instruction, but this instruction is always preceded by a save of the %pc to be able to return. Nevertheless, our definition does not handle properly the *tail call* optimization, because calls do not include a push of a return address in this kind of scenario. We argue that, at binary level, it is reasonable to consider this kind of pattern is not a function call. The only reason we may want to include them is to comply with the source code it comes from. We propose heuristics in Section 8.2 in order to include tail calls to our definition and thus to retrieve function calls as they were written at source level.

5.2.2 Memory location and parameters

In this section, we propose a definition for parameters and return values of functions. First, we define the memory locations where parameters can hold, and second we propose a definition of parameters and return values.

5.2.2.1 Memory location of parameters and return values

We propose here two sets of memory locations that we use then to define parameters and return values of functions. In Definition 5, we defined the memory locations of an execution e. From this, we define two subsets of $\mathcal{M}^*(e)$, where functions are susceptible of getting parameters/passing return values.

Definition 12. *We define the parameter memory locations, relatively to an execution e, denoted as $\mathcal{M}_p(e)$, as the subset of $\mathcal{M}^*(e)$ where any function f can read a value set before f was called.*

Definition 13. *We define the return value memory locations, relatively to an execution e, denoted as $\mathcal{M}_r(e)$, as the subset of $\mathcal{M}^*(e)$ where any function f can write a value that will be read after t returns.*

Note that one can arbitrary set $\mathcal{M}_p(e)$ and $\mathcal{M}_r(e)$, according to a given calling convention for example. In the general case, one can set $\mathcal{M}_p(e) = \mathcal{M}_r(e) = \mathcal{M}^*(e)$.

5.2.3 Parameter and return value

As a part of our dynamic approach, we define in the first place a notion of parameter (and return value) that is relative to a given execution ε of a function f. In Section 5.2.4, we propose a definition of *arity* that generalizes it and that is independent from a given execution of f.

5.2.3.1 Definition

We define a parameter of f as a location of $\mathcal{M}$ that is accessed in a particular way by f. Note that, according to what memory location is included or not into $\mathcal{M}$, this definition will include or not different entities as parameters.

Definition 14. *A parameter of f, relatively to an execution $\varepsilon \in e\lfloor f$, is a memory location $m \in \mathcal{M}_p(e)$ such that:*

1. *the content of m is* **read before written** *by f during ε,*
2. *the value of m (i.e., the address of the memory location) is* **not** *computed from a $m' \in \mathcal{M}_p(e)$ read before written by f during ε.*

In the same way, we define the notion of return values, in the frame of an execution ε of f, according to $\mathcal{M}_r$.

Definition 15. *A return value of f, relatively to an execution $\varepsilon \in e\lfloor f$, is a memory location of $\mathcal{M}_r(e)$ such that:*

1. *the content of m is written by f during ε,*
2. *the content of m is read (before written) after ε, i.e., after f has returned,*
3. *the value of m (i.e., the address of the memory location) is* **not** *computed from a $m' \in \mathcal{M}_r(e)$ written by f during ε and read after ε.*

5.2.3.2 Discussion

We propose to discuss in this section several points of our definitions.

Parameter We consider in this definition that a parameter is a location that is read before being written by a function f. This location can either be a register or a memory address. For instance in x86 architectures, if f performs a save of the register `%ebp`[1]: the value of `%ebp` will thus be read before being written and so will be considered as a parameter of f, as long as `%ebp` is in $\mathcal{M}_p(e)$. However, we introduce a limitation with the second criterion (*"not computed from..."*). Locations corresponding to registers are addressed directly and never require a computation ; but other memory locations may need a computation from a base address, plus an offset (for instance). What we are saying with this criterion is that a memory location obtained from a parameter should not be considered as another (new) parameter. If f takes an address a as a parameter, and if it accesses $a + 4$, the memory location $a + 4$ should not be considered as another parameter of f, even if its content is read before written.

In other words, parameters are memory locations, directly addressable[2] *by f, that are read before written.*

Return value According to the definition we propose, a return value is a location that is written by f and that is reused after f has returned. Functions usually use registers to perform computation, to store for example intermediate results. They also push parameters on the stack and may access other memory location. However, these are not return values of f as long as they are not used further in the execution. In addition, we use the same argument as for parameters to exclude memory locations computed from a return value (*"not computed from..."*).

In other words, return values are memory locations, directly addressable by the caller, that are written by f and read by one of its callers.

5.2.3.3 Examples

Parameter We propose several examples to illustrate, in different common cases, what would be considered as a parameter according to Definition 14 and what would not.

Register parameter In Listing 5.1, the value of registers `%rbp`, `%rsp` and `%edi` are read before being written. Therefore, if $\mathcal{M}_p(e)$ contains all registers, then

[1] This register, in x86 architectures, stores the address of the base of the stack for the current function

[2] *i.e.*, using a direct addressing mode

this function takes at least these three parameters. On the other hand, one can choose to exclude %rbp and %rsp from $\mathcal{M}_p(e)$, in which case the only remaining parameter is %edi. Note that the location %rbp - 0x14 is written before being read, thus it is not a parameter of f.

```
00000000004004ad <f>:
  4004ad:       55                      push   %rbp
  4004ae:       48 89 e5                mov    %rsp,%rbp
  4004b1:       89 7d ec                mov    %edi,-0x14(%rbp)
  4004b4:       8b 45 ec                mov    -0x14(%rbp),%eax
  4004b7:       83 c0 01                add    $0x1,%eax
  4004ba:       89 45 fc                mov    %eax,-0x4(%rbp)
  4004bd:       8b 4d fc                mov    -0x4(%rbp),%ecx
   ...:   ...           ...     ...
```

Listing 5.1: Example of register parameter

Stack parameter In Listing 5.2, we have %ebp, %esp and %ebp + 0x8 that are read before written. Depending on the set $\mathcal{M}_p(e)$, we end up with different results. *If* %esp *and* %ebp *are in* $\mathcal{M}_p(e)$, then they both are considered as parameters. However, the location %ebp + 8 is computed from %ebp in $\mathcal{M}_p(e)$ which is read before written, so it would not be considered as a parameter of f. *If* %ebp *is not in* $\mathcal{M}_p(e)$, then it will not be considered as a parameter, but %ebp + 8 will be. This later case is more compliant with the notion of parameters at source level - this will be discussed in Section 8.2.

```
080483cb <f>:
 80483cb:       55                      push   %ebp
 80483cc:       89 e5                   mov    %esp,%ebp
 80483ce:       83 ec 10                sub    $0x10,%esp
 80483d1:       8b 45 08                mov    0x8(%ebp),%eax
 80483d4:       83 c0 01                add    $0x1,%eax
 80483d7:       89 45 fc                mov    %eax,-0x4(%ebp)
 80483da:       8b 4d fc                mov    -0x4(%ebp),%ecx
   ...:   ...           ...     ...
```

Listing 5.2: Example of stack parameter

Global variable Listing 5.3 present an example of a function that access a global variable: the location 0x80497a0 is read before being written, and is not

computed from another memory location read before written. Therefore, this location will be considered, according to Definition 14, as holding a parameter of f.

```
08048422 <f>:
 8048422:       55                      push   %ebp
 8048423:       89 e5                   mov    %esp,%ebp
 8048425:       83 ec 10                sub    $0x10,%esp
 8048428:       a1 a0 97 04 08          mov    0x80497a0,%eax
 804842d:       89 45 fc                mov    %eax,-0x4(%ebp)
   ...:   ...           ...     ...
```

Listing 5.3: Example of global variable

Address In Listing 5.4, %ebp, %esp, %ebp + 8, *(%ebp + 8) and *(%ebp + 8) + 4 are memory locations that are read before being written. If we exclude %ebp and %esp from $\mathcal{M}_p(e)$, we have %ebp + 8 which is a parameter, that we denote by a. Then, the function f accesses the memory location pointed by a (*i.e.*, *a), and then the memory location pointed by $a + 4$. Because a is read before written, any memory location computed from a is not considered as a parameter, and thus $*a$ and $*(a + 4)$ are not parameters of f.

```
0804843d <f>:
 804843d:       55                      push   %ebp
 804843e:       89 e5                   mov    %esp,%ebp
 8048440:       83 ec 10                sub    $0x10,%esp
 8048443:       8b 45 08                mov    0x8(%ebp),%eax
 8048446:       8b 00                   mov    (%eax),%eax
 8048448:       89 45 fc                mov    %eax,-0x4(%ebp)
 804844b:       8b 45 08                mov    0x8(%ebp),%eax
 804844e:       8b 40 04                mov    0x4(%eax),%eax
 8048451:       89 45 f8                mov    %eax,-0x8(%ebp)
   ...:   ...           ...     ...
```

Listing 5.4: Example of a parameter used to access memory locations

Return value We propose here three concrete examples of return value to illustrate our definition.

Through register We give in Listing 5.5 an example of a return value passed through a register (in this case, %eax). At address 0x80484e1, the content of

the register %eax is set by f. At address 0x80484f0, f is called, and at address 0x80484f8, *i.e.* after f has returned, the content of %eax is read by *main*. This corresponds to a return value of f, in the case where %eax is in $\mathcal{M}_r(e)$.

```
080484cd <f>:
     ...:       ...                     ...     ...
 80484e1:       b8 00 00 00 00          mov     $0x0,%eax
 80484e6:       c9                      leave
 80484e7:       c3                      ret

080484e8 <main>:
     ...:       ...                     ...     ...
 80484f0:       e8 d8 ff ff ff          call    80484cd <f>
 80484f5:       83 c4 04                add     $0x4,%esp
 80484f8:       89 45 fc                mov     %eax,-0x4(%ebp)
     ...:       ...                     ...     ...
```

Listing 5.5: Example of a return value passed through register

Out parameter In Listing 5.6, we propose an example of what is commonly called an *out parameter*, *i.e.* a parameter that is a location where to write a result. In this example, f takes an address, that we will denote as a, as a parameter. It corresponds to the location %ebp + 8. At address 0x80484fa, it writes the content of %edx to this location a. Then, although is is not explicit statically, the function *main* reads this value at address 0x804851e[3]. Consequently, this location a is, according to Definition 15, a return value of f (if it is included in $\mathcal{M}_r(e)$).

```
080484e8 <f>:
 80484e8:       55                      push    %ebp
 80484e9:       89 e5                   mov     %esp,%ebp
     ...:       ...                     ...     ...
 80484f7:       8b 45 08                mov     0x8(%ebp),%eax
 80484fa:       89 10                   mov     %edx,(%eax)
 80484fc:       90                      nop
 80484fd:       c9                      leave
 80484fe:       c3                      ret

080484ff <main>:
 80484ff:       55                      push    %ebp
 8048500:       89 e5                   mov     %esp,%ebp
 8048502:       83 ec 10                sub     $0x10,%esp
 8048505:       6a 00                   push    $0x0
 804850c:       83 c4 04                add     $0x4,%esp
```

[3]In fact, we can deduce statically that %ebp + 8 in f corresponds to the same location as %ebp - 4 in *main* ; however this is explicit in a dynamic analysis context

```
8048515:        50                      push   %eax
    ...:        ...                     ...    ...
8048516:        e8 cd ff ff ff          call   80484e8 <f>
804851b:        83 c4 04                add    $0x4,%esp
804851e:        8b 45 fc                mov    -0x4(%ebp),%eax
8048521:        c9                      leave
8048522:        c3                      ret
    ...:   ...           ...    ...
```

Listing 5.6: Example of an "out parameter", *i.e.* a return value passed by reference

Global variable Listing 5.7 presents a function that uses a global variable to return a value (in this case, the global variable is at address 0x8049900. At address 0x8048505, this global variable is written by f ; and at address 0x8048538 it is read by `main` after f has returned. Here again, if $0x80048505 \in \mathcal{M}_R(e)$, then it will be considered as a return value of f.

```
080484ff <f>:
    ...:   ...           ...    ...
 8048505:       a3 00 99 04 08          mov    %eax,0x8049900
 804850a:       90                      nop
 804850b:       5d                      pop    %ebp
 804850c:       c3                      ret

0804850d <main>:
    ...:   ...           ...    ...
 8048530:       e8 ca ff ff ff          call   80484ff <f>
 8048535:       83 c4 04                add    $0x4,%esp
 8048538:       a1 00 99 04 08          mov    0x8049900,%eax
    ...:   ...           ...    ...
```

Listing 5.7: Example of a global variable used to return a value

5.2.3.4 Consistency with `C` compiled binaries

In this section, we show that Definition 14 and Definition 15 are compliant with the notion of parameters and return values at source level in `C`, if $\mathcal{M}_p(e)$ is wisely chosen. In fact, every example given in the previous section was obtained by compilation from `C` ; we propose to make explicit the set of $\mathcal{M}_p(e)$ for each example that leads to obtain parameters and return values in keeping with the `C` source code.

Parameter

Register and stack parameter Listing 5.8 is the C code from which Listings 5.1 and 5.2 were obtained (one was compiled for 32bit architecture, the other one for 64bit architecture). In this source code, f has one parameter (namely a). With $\mathcal{M}_p(e)$ that excludes %ebp and %esp, in both cases from the assembly code we retrieve this one parameter (%edi in the first case and %ebp + 0x8 in the later case).

```
int f(int a) {
    int b = a + 1;
  ...
```

Listing 5.8: C code of a function with one parameter - source from which Listings 5.1 and 5.2 were obtained

Global variable Listing 5.3 corresponds to a compilation of the C code presented in Listing 5.9. In this scenario, the location of the global variable (here 0x80497a0) is considered as a parameter of f if $\mathcal{M}_p(e)$ includes any addressable memory location. However, to be compliant with source level, *i.e.* to exclude VALUE from the parameters of f, one can set $\mathcal{M}_p(e)$ in a way that excludes the data segment of the binary under analysis. Thus, f would have no parameter according to our definition, which is what is defined at source level.

```
int VALUE = 0xcafe;

int f(void) {
    int a = VALUE;
  ...
```

Listing 5.9: C code of a function using a global variable - source from which Listing 5.3 was obtained

Return value In C, a function can return either none or one parameter, but no more. This means that, to be compliant with the prototypes of functions as defined in C, our definition of a return value should exclude out parameters and global variables (at least). To exclude global variables, it is enough to set $\mathcal{M}_r(e)$ to exclude the data segment of the binary under analysis, where global variables are

stored. Regarding out parameters, we could add a criterion to exclude from $\mathcal{M}_r(e)$ every location that is computed from input parameters of f.

5.2.4 Arity

5.2.4.1 Inputs

For each execution ε of f, we denote by $in_\varepsilon(f)$ the set of $m \in \mathcal{M}(f)$ that are parameters of f during ε, according to Definition 14. We denote by $in_e(f)$ the set of $m \in \mathcal{M}(f)$ that are parameters of f during e:

$$in_e(f) = \bigcup_{\varepsilon \in e \lfloor f} in_\varepsilon(f)$$

In other words, $in_e(f)$ is the set of memory locations that are detected as parameters of f during at least one execution of f.

By extension, we define the set of parameters of f relative to any possible execution e:

Definition 16. *The set of parameters of $f \in \mathcal{F}(\mathcal{E}(B))$ is*

$$in(f) = \bigcup_{e \in \mathcal{E}(B)} in_e(f)$$

From this, we can define the notion of arity of f relatively to parameters as follows.

Definition 17. *The parameter arity of f, denoted by $\#_p f$, is the number of different memory locations of $\mathcal{M}_p$ that are parameters in at least one possible execution ε of f.*

$$\#_p f = |in(f)|$$

We denote by $\#_p f_e = |in_e(f)|$ the parameter arity of f relative to one execution e of the binary under analysis B.

5.2.4.2 Outputs

In the same way, we denote for each execution ε of f $out_\varepsilon(f)$ the set of $m \in \mathcal{M}_r$ that are return values according to Definition 15, and the set of return values of f during e:

$$out_e(f) = \bigcup_{\varepsilon \in e \lfloor f} out_\varepsilon(f)$$

We define by extension the set of return values of f relatively to any possible execution e:

Definition 18. *The set of return values of $f \in \mathcal{F}(\mathcal{E}(B))$ is*

$$out(f) = \bigcup_{e \in \mathcal{E}(B)} out_e(f)$$

From this, we define the return value arity of f as follows.

Definition 19. *The return value arity of f, denoted by $\#_r f$, is the number of different memory locations of $\mathcal{M}_r$ that are return values in at least one possible execution ε of f:*

$$\#_r f = |out(f)|$$

We denote by $\#_r f_e = |out_e(f)|$ the return value arity of f relative to one execution e of the binary under analysis B.

5.2.4.3 Order of parameters

We propose to order the two sets $in(f)$ and $out(f)$ with indexes from 1 to $\#_p f$ and from 1 to $\#_r f$ respectively, so we can designate parameters of f by their index. For any $i \in [\![1, \#_p f]\!]$, we denote by $f_p[i]$ the memory location holding the i^{th} parameter of f, and for any $j \in [\![1, \#_r f]\!]$, we denote by $f_r[j]$ the j^{th} return value of f, accordingly to a given order of the two sets. This order can be set, for example, regarding the calling convention (they usually specify an order of memory locations to be used to pass parameters). In any case, this order should be invariant, and should not depend on the considered execution. We may not detect the first and third parameter of a function during an execution e, but the second parameter would still be indexed by 2.

5.2.5 Type

In this section, we propose definitions relative to typing of parameters. First, we present what we aim to express through the notion of typing, and then we propose definitions of *operations* and *typing*.

5.2.5.1 Nature

Typing is usually considered as a high-level notion, although it also exists at binary level. The main difference is that, whereas at source level, type is often explicitly determined (either set or inferred), at binary level there is no meta-data associated to a memory location to store its type. Typing is then relative to operations that can be performed on a data, and not defined explicitly when the data is set. For instance, an address is not explicitly declared as such in a binary code, however it will probably be used to access a memory location, whereas an integer value would not. Usually, the notion of type of a variable holds in fact three levels of information:

1. the nature of data (address, integer, etc.) from a semantic point of view,
2. the size of data (one byte, two bytes, four bytes, etc.),
3. meta-data (nature of the data pointed by an address, signed or unsigned integer, etc.).

At source level, these three levels are useful, for instance to perform type checking (usually at compilation time) and to optimize the assembly code produced. However, the semantic distinction between variables is made at level 1 only : the kind of data defines the operations that can be performed on the variable, whereas the size may not.

In this work, we call typing the nature of the data, i.e., according to the kind of operations that can be performed on it. The following definitions lead to a precise description of *typing* regarding function parameters and return values.

5.2.5.2 Operations

Instructions usually manage their operands asymmetrically. Therefore, we introduce the notion of operation, as a couple of an instruction and one of its operands.

Definition 20. *Given an instruction ins = (opcode, operands), we construct* $|opcode|$ *operations as follows:*

$$(opcode, operands) \longrightarrow \{(opcode, i), \forall i \in [\![1, |opcode|]\!]\}$$

We denote by $(op, i)(p)$ the operation (op, i) applied to the parameter p.

5.2.5.3 Typing

For each parameter of a given function, we define its type according to the kind of operations that can be performed on it. We first define the notion of set of possible types, which corresponds to the types that are considered in a given context.

Definition 21. *A type t is defined by a set of operations (op, i) that can be applied to data of this type.*

Definition 22. *The set of types $\mathcal{T}(e)$ is the set of types that are considered during an execution e.*

In other words, $\mathcal{T}$ is a set of sets of operations.

Definition 23. *The set of operations, during an execution e, used to discriminate types, is the set of all operations defining at least one type:*

$$\mathcal{O}(e) = \bigcup_{t \in \mathcal{T}} \{(op, i) \in t\}$$

From these definitions, we can express the function *type* for parameters of a function, according to a given execution.

Definition 24. *For a function f, an execution $\varepsilon \in e\lfloor f$, and for any $p \in in_\varepsilon(f) \cup out_\varepsilon(f)$, we define*

$$type_\varepsilon(p) = \mathcal{T}(e) \setminus \{t \in \mathcal{T}(e) \mid \exists (op, i) \in \mathcal{O}(e), (op, i)(p) \text{ and } (op, i) \notin t\}$$

This formula states that possible types of p given the execution ε is the set of all considered types for which we have not observed inconsistency, *i.e.* discriminating operations of other types applied to p.

Note that any operation performed on p that is not in $\mathcal{O}(e)$ has no influence on its type. We generalize Definition 24 to the entire execution e:

Definition 25. *For a function f and an execution e, we define the possible types of $p \in in(f) \cup out(f)$:*

$$type_e(p) = \bigcap_{\varepsilon \in e\lfloor f} type_\varepsilon(p)$$

From this definition of possible types, three scenarios can occur:

1. $type_e(p) = \emptyset$: this means that the set of operations applied to p does not correspond to any type $t \in \mathcal{T}(e)$. In this case, we conclude that the type of p is inconsistent (noted `INCONS`).

2. $|type_e(p)| > 1$: this means that the set of operations applied to p is not enough to conclude on the type of p: several types are consistent with the operations seen during e. In this case, we conclude that the type of p is undefined (noted `UNDEF`).

3. $|type_e(p)| = 1$: exactly one type t of $\mathcal{T}(e)$ is consistent with the set of operations performed on p. In this case, p is typed with t.

Finally, we define the type of a parameter (resp. return value) regarding any execution as follows:

Definition 26. *For a function f of a binary B, let us denote by S_t the set*

$$S_t = \bigcap_{e \in \mathcal{E}(B)} type_e(p)$$

for $p \in in(f) \cup out(f)$. We define the type of p as follows:

$$type(p) = \begin{cases} \text{the unique element of } S_t \text{ if } |S_t| = 1 \\ \texttt{UNDEF} \text{ if } |S_t| > 1 \\ \texttt{INCONS} \text{ if } |S_t| = 0 \end{cases}$$

This definition is a generalization of Definition 25 to every execution of B.

5.2.5.4 Example

A concrete instance of these definitions to propose a typing is given in Section 6.2.1, where we present our approach. In addition, we propose here a basic example with three considered types: booleans, integers and floats. We define the sets corresponding to each type as follows:

- $bool = \{\texttt{AND}, \texttt{OR}, \texttt{XOR}, \texttt{NOT}, \texttt{CMP}\}$,
- $int = \{\texttt{ADD}, \texttt{SUB}, \texttt{MULT}, \texttt{XOR}, \texttt{DIVE}^4, \texttt{CMP}, \texttt{MOD}\}$,

[4] Euclidean division

- $float = \{\texttt{ADD}, \texttt{SUB}, \texttt{MULT}, \texttt{DIVF}^{5}, \texttt{CMP}\}$.

Note that in this instruction set, operations are not typed, except for the division (`DIVE` and `DIVF`). For example, the `ADD` instruction can be applied to a float or an integer. In some instruction sets, there would be different instructions to deal with different types of variable.

Now consider in Listing 5.10 three executions of the same function f, taking one parameter $f_p[1]$ at memory location `%rdi`, from which we want to retrieve the type. The given instructions are the one actually executed during the call of f.

```
// First execution of f
f:
    PUSH    %ebp
    MOV     %esp, %ebp
    SUB     0x4, %esp
    MOV     %edi, -0x4(%ebp)
    CMP     -0x4(%ebp), 0
    JLE     ret_err
ret_err:
    MOV     -1, %eax
    RET
// Second execution of f
f:
    PUSH    %ebp
    MOV     %esp, %ebp
    SUB     0x4, %esp
    MOV     %edi, -0x4(%ebp)
    CMP     -0x4(%ebp), 0
    JLE     ret_err
    CMP     -0x4(%ebp), 0xFF
    JGE     mask
    SUB     1, _-0x4(%ebp)
    CMP     -0x4(%ebp), 0
    JE      ret_err
ret_err:
    MOV     -1, %eax
    RET
// Third execution of f
f:
    PUSH    %ebp
    MOV     %esp, %ebp
    SUB     0x4, %esp
    MOV     %edi, -0x4(%ebp)
    CMP     -0x4(%ebp), 0
    JLE     ret_err
    CMP     -0x4(%ebp), 0xFFFF
    JGE     mask
mask:
```

[5] Floating division

```
MOV     -0x4(%ebp), %eax
XOR     0xFFFF, %eax
...
```

Listing 5.10: Three executions of a function f taking one parameter

During the first execution ε_1 of f, the only operation performed on $f_p[1]$ is a comparison (`CMP`), which does not allow to conclude. The second execution ε_2 leads to more instructions to be executed, and in particular instruction `SUB`. Finally, during the third execution ε_3 of f, instructions `CMP` and `XOR` are computed.

According to Definition 24, and $\mathcal{O}(e)$ deduced from the instructions specified in definitions of $bool$, int and $float$, we have:

$$\begin{cases} type_{\varepsilon_1}(f_p[1]) = \{bool, int, float\} \\ type_{\varepsilon_2}(f_p[1]) = \{int, float\} \\ type_{\varepsilon_3}(f_p[1]) = \{bool, int\} \end{cases}$$

From these three sets, we can deduce the type of the first parameter of f in the execution e:

$$type_e(f_p[1]) = \bigcap_{\varepsilon \in e \lfloor f} type_\varepsilon(f_p[1]) = \{int\}$$

In this example, the intersection of these sets gives a set with exactly one element, so we can conclude that $f_p[1]$ is typed with the type int.

5.3 Behavior

The previous section was related to structure of a binary, and focuses on the functions themselves (and in particular its parameters and return values). In this section, we define notions relative to the behavior of a binary program during its execution. Our point of view is still at the granularity of functions, so the behavior we aim to capture is relative to the way functions are called during the execution. We also consider data flow to express a sharing of information between functions.

5.3.1 Number of calls

First, we introduce a notation relative to the number of calls of functions. For any execution e, and for any function $f \in \mathcal{F}(e)$, we denote by $nc_e(f)$ the number of

times f is called during execution e. For function $f \in \mathcal{F}(e)$ that is never executed during e, $nc_e(f) = 0$.

In other words, $nc_e(f)$ is the number of times %pc *takes as a value the address of the first instruction of f during e.*

5.3.2 Data flow

We need to define a notion of data flow from a memory location m to another memory location m'.

Definition 27. *Given two memory locations $m_1 \in M^*(e)$ and $m_2 \in M^*(e)$ and two instructions i_1 and i_2 of an execution e, there is a data flow between (i_1, m_1) and (i_2, m_2) if:*

- *the minimum content of m_1 is copied to location m_2 at instruction i_c between instructions i_1 and i_2 and the minimum content of m_2 is* **not modified** *in any way between i_c and i_2,*
- *or there exists $m_3 \in M^*(e)$ and an instruction i_3 such that there is a data flow between (i_1, m_1) and (i_3, m_3) and between (i_3, m_3) and (i_2, m_2).*

We denote a data flow between (i_1, m_1) and (i_2, m_2) by $(i_1, m_1) \rightharpoonup (i_2, m_2)$.

Definition 28. *Given two memory locations $(m, m') \in M^*(e)^2$, there is a rupture of data flow between m and m' at instruction i_r, if there exists an instruction $i_0 < i_r$ such that $(i_0, m) \rightharpoonup (i_r - 1, m')$ and the minimum content of m' is modified at instruction i_r (where $i_r - 1$ is the instruction that immediately precedes i_r).*

5.3.3 Def-use

5.3.3.1 Def-use chain

We also define a *def-use* chain between two functions f and g of $\mathcal{F}(e)$, as follows:

Definition 29. *For $f \in \mathcal{F}(e)$ and $g \in \mathcal{F}(e)$ two functions of the execution e, there is a def-use chain between f and g if:*

- *f writes a value at a memory location $m \in \mathcal{M}_r(e)$ during an execution $\varepsilon_f \in e\lfloor f$ at instruction i,*

- *g reads a value at a memory location* $m' \in \mathcal{M}_p(e)$ *during an execution* $\varepsilon_g \in e\lfloor g$ *posterior to* ε_f*, at instruction* i'*,*
- *there is a data flow between* (i, m) *and* (i', m')*.*

In other words, there is a def-use chain between f and g when f sets a value that is used later by g.

Example Listing 5.11 presents an illustration of a def-use chain between f and g. First, f is called and outputs a value through the register `%eax`. This output value is stored on the stack, and later (with no overriding in between) it is passed to f through register `%edi`[6].

```
40059f:       bf 28 00 00 00          mov    $0x28,%edi
4005a4:       e8 40 01 00 00          callq  4006e9 <f>
4005a9:       48 89 45 f8             mov    %eax,-0x8(%rbp)
...    :       // -0x8(%ebp) is not overriden in between
4005c3:       48 8b 45 f8             mov    -0x8(%rbp),%eax
4005c7:       be 28 00 00 00          mov    $0x28,%esi
4005cc:       48 89 c7                mov    %eax,%edi
4005cf:       e8 22 04 00 00          callq  4009f6 <g>
```

Listing 5.11: Example of a def-use chain - static illustration

5.3.3.2 Def-use value

From this definition, we define a *def-use value* between a parameter and a return value.

Definition 30. *For* $f \in \mathcal{F}(e)$ *and* $g \in \mathcal{F}(e)$ *two functions of the execution e, and for* $i \in [\![1, \#_r f]\!]$ *and* $j \in [\![1, \#_p g]\!]$*, we define a def-use value between* $f_r[i]$ *and* $g_p[j]$*, denoted by* $df_e(f_r[i], g_p[j])$*, the number of times a def-use chain exists between f and g such that the memory location written by f is* $f_r[i]$ *and the memory location read by g is* $g_p[j]$ *during e.*

In other words, the def-use value of $f_r[i]$ and $g_p[j]$ is the number of times the i^{th} return value of f is used as a j^{th} parameter of g.

[6] In x86-64 System-V calling convention, the register `%eax` is used by functions to return values during a return; `%edi` and `%esi` are the two registers used to store the first two parameters being passed during a call.

5.3.4 Coupling

Now, let f and g be two functions of $\mathcal{F}(e)$, for e a given execution of the binary B. We define the *ρ-coupling* as follows:

Definition 31. *For $i \in [\![1, \#_r f]\!]$ and $j \in [\![1, \#g]\!]$, $f_r[i]$ and $g[j]$ are ρ-coupled if:*

- $type(f_r[i]) = type(g_p[j])$,
- $df_e(f_r[i], g_p[j]) \geq \rho \times nc(g, e)$.

The value of ρ is called the rate of a coupling between $f_r[i]$ and $g_p[j]$.

In other words, $f_r[i]$ and $g_p[j]$ are ρ-coupled if they have the same type, and the j^{th} parameter values of g during the execution e come from the i^{th} return value of f in a proportion of at least ρ.

5.3.4.1 Discussion

We introduce the typing in our definition of coupling to add a semantic meaning to the def-use chain. We could imagine def-use chains between an allocator and a random generator for instance, where the random generator would use addresses as seeds. In this case, there would be a def-use chain between the two, but we exclude this in our definition because the use of this value would be semantically very different in an allocator and in a random generator.

5.4 Allocator

One can find, for example in [CSB13], empirical definitions of allocators, mostly based on observation and heuristics. In this section, we attempt to propose a general definition of what is an allocator of resources, independently from the kind of resources we deal with. The goal is to understand what we want to retrieve in Chapter 7, and to justify heuristics and approximations we make. In the rest of this work, we apply these definitions in the specific context of memory allocators (see Section 7.2).

5.4.1 Resource

First of all, we define a resource as an indexed set of elements that can be used.

Definition 32. *A resource $\mathcal{R}$ of size M is a set of M "cells", each one denoted by an index $p \in \mathbb{N}$:*

$$\mathcal{R} = \{p_i \in \mathbb{N}, i \in [\![0, M-1]\!]\}$$

In other words, $\mathcal{R}$ is represented by a subset of $\mathbb{N}$, of size M.

A memory space indexed by addresses is one example of resource, as well as time slots or a set of `CPUs` that can be assigned tasks.

From a resource, we can define subsets of the resource R, that we denote by $r \subset R$.

5.4.2 Block

Given a resource R composed of indexed cells, we can define blocks of cells as a sequence of consecutive cells of R.

Definition 33. *A resource block is a sequence of consecutive cells, denoted by $|p, s\rangle$ with $(p, s) \in \mathbb{N} \times \mathbb{N}$:*

$$|p, s\rangle = \{p + i, i \in [\![0, s-1]\!]\}$$

For any $p \in \mathbb{N}$, we have $|p, 0\rangle = \{\emptyset\}$. We denote by $\mathcal{B}$ the set of all resource blocks.

Definition 34. *A block (p, s) is a resource block over $\mathcal{R}$ if and only if*

$$\forall i \in [\![0, s-1]\!], p + i \in \mathcal{R}$$

We denote by $\mathcal{B}(\mathcal{R})$ the set of resource blocks of $\mathcal{R}$.

5.4.3 Fragmentation

A set of blocks that do not overlap also define a resource. We call it fragmented resource.

Definition 35. *We define the set of fragmented resources, denoted by P, in the following way:*

$$\forall \pi \in 2^{\mathbb{N}\times\mathbb{N}}, \pi \in P \Leftrightarrow \forall (x, y) \in \pi \times \pi, x \neq y \Rightarrow |x\rangle \cap |y\rangle = \{\emptyset\}$$

In other words, a fragmented resource is a resource described as a set of blocks that do not overlap.

A given resource r can always be described in at least one fragmented way.

Definition 36. *For $r = \{p_i, i \in [\![0, M-1]\!]\}$, a fragmented description of r is*

$$\pi = \{(p_i, 1), i \in [\![0, M-1]\!]\}$$

Note that this fragmented representation is not unique.

5.4.4 Projection

To explicit the link between fragmented resources and the set of cells they are defined over, we introduce the projector operator.

Definition 37. *For a fragmented resource $\pi \in P$, we call support or projected form and we denote by $\pi\downarrow$ the resource composed by every cell of π :*

$$\pi\downarrow \ = \{p + j, (p, s) \in \pi \text{ and } j \in [\![0, s-1]\!]\}$$

5.4.5 Fragmented state of a resource

In the other way, we can define the set of possible fragmented descriptions of a given resource from its projected form.

Definition 38. *For a given resource state $r \in \mathcal{R}$, the set of all possible fragmented descriptions $P(r)$ of r is:*

$$P(r) = \{\pi \in P \mid \pi\downarrow = r\}$$

Definition 39. *By extension, for a resource $\mathcal{R}$, we define $P(\mathcal{R})$ as the set of all possible fragmented descriptions of any state r of $\mathcal{R}$:*

$$P(\mathcal{R}) = \{\pi \in P \mid \exists r \subset \mathcal{R}, \pi\downarrow = r\}$$

5.4.6 Allocator of resources

Now that we have defined a resource and its possible fragmented descriptions, we can define an allocator of resource.

5.4.6.1 Allocated resource, resource available

Let $\mathcal{R}$ be the addressable resource for a given program. We define $\pi \in P(\mathcal{R})$ as the resource allocated by the program at a given point of the execution. This allocated resource is described in a fragmented form. π induces two subsets of $\mathcal{R}$ that are complementary:

- $\pi\downarrow \subset \mathcal{R}$: support of the fragmented resource allocated by the program
- $\overline{\pi\downarrow} \subset \mathcal{R}$: resource not allocated by the program

A resource cell cannot be allocated and not allocated in the same time.

5.4.6.2 Allocate and free

Definition 40. *We define an allocating function $alloc$ as follows:*

$$alloc \left| \begin{array}{lll} P(\mathcal{R}) & \longrightarrow & P(\mathcal{R}) \\ \pi & \longrightarrow & \pi' \end{array} \right.$$

such that:

$$\pi' = \pi \cup \{(p,s)\}, (p,s) \in \mathcal{B}(\overline{\pi\downarrow})$$

The block (p,s) added to π' can be the empty block ($s = 0$), for instance in the case of a failure during the allocation.

Definition 41. *We define a freeing function $free$ as follows:*

$$free \left| \begin{array}{lll} P(\mathcal{R}) & \longrightarrow & P(\mathcal{R}) \\ \pi & \longrightarrow & \pi' \end{array} \right.$$

such that:

$$\forall (p',s') \in \pi', \exists (p,s) \in \pi, \{(p',s')\}\downarrow \subset \{(p,s)\}\downarrow$$

Remarks. Every block in π' (after being freed) is included in a block of π (present before the call to free). In particular:

- a free **cannot** bring new resource cells in π',

- a free **cannot** lead to a new organization of allocated blocks (*e.g.* merge of two consecutive blocks),
- a free **can** target a subpart of a block (partial free),
- a free **can** target several blocks (multiple free).

5.4.6.3 Allocator

Definition 42. *We define an allocator of a given resource r as a couple $(alloc, free)$ of an allocating function $alloc$ and a freeing function $free$, both defined over R.*

Chapter 6

Structure inference

As presented in Chapter 2, structure-level information is interesting to retrieve from a binary, in a reverse-engineering context. In this chapter, we propose to retrieve structure at a function-level granularity. In particular, we are focusing on the arity of functions and the type of their parameters.

We assume, as a starting point, that functions are well-identified in the binary. Although we discussed about the difficulties to retrieve functions in a binary, several works have been conducted (see [BBW+14] and [ASB17] for instance) with good success. In addition, our implementation, we rely on `Pin` to detect the beginning and the end of routines. Retrieving functions is thus not in the scope of this work.

6.1 Arity

6.1.1 Problem

The problem of arity we address in this chapter is the following:

Problem 1. *Given a binary program $\mathcal{B}$, for any $f \in \mathcal{F}(\mathcal{E}(\mathcal{B}))$, we want to retrieve the two sets $in(f)$ and $out(f)$.*

In other words, we aim to retrieve the parameters and return values of any function f executed during e. Usually, *arity* corresponds to the number of parameters a function take, *i.e.* in our case, it would mean $\#_p f$ and $\#_r f$. However, we assume that we do not know where parameters are located. Thus, our notion of *arity* includes the location of parameters.

6.1.2 Criteria

In Section 4.1.3, we presented the three main criteria we focus on in this work. The *accuracy* criterion will be evaluated through experiments (see Chapter 9). However, we propose definitions of false positive and false negative in order to evaluate the kind of errors our approach may encounter. We also propose discussions about *scalability* and *universality*.

6.1.2.1 Accuracy

To discuss the accuracy criterion, we first propose definitions of what we call *false positive* and *false negative*, relatively to Problem 1.

Definition 43. *We call false positive in an execution e, a memory location $m \in \mathcal{M}_p(e) \setminus in_e(f)$ (resp. $m \in \mathcal{M}_r(e) \setminus out_e(f)$) that is detected as a parameter (resp. return value) of f.*

In other words, a false positive corresponds to detecting a parameter that does not exist.

Definition 44. *We call false negative in an execution e, a memory location $m \in in_e(f)$ (resp. $m \in out_f(f)$ that is not detected as a parameter (resp. return value) of f.*

In other words, a false negative corresponds to not detecting some parameters that exist. To comply with the criterion of accuracy we mentioned in Section 4.1.4, we aim to limit both false positives and false negatives.

6.1.2.2 Universality

Passing parameters (and returning values) can be performed through many mechanisms. In well known architectures (`x86`, `x86-64`, `ARM`,...), it is either through the stack or through registers (and sometimes both) ; but one can imagine more esoteric methods. The flexibility of the approach we propose relies on the possibility to define $\mathcal{M}_p(e)$ and $\mathcal{M}_r(e)$, the two sets that specify the memory locations that might be used to pass parameters and to return values. They can be defined for each binary $\mathcal{B}$, depending on the architecture, the calling convention being used, etc. ; and for each execution e, depending on the execution parameters. However, some cases cannot be handled by this method:

1. if the calling convention is unknown: this can happen if the program header is incorrect or if the calling convention is not documented *e.g.*, a custom calling convention,

2. if functions use different calling conventions,

3. if the calling convention changes during the execution.

In the last two cases, one single definition of $\mathcal{M}_p(e)$ (and $\mathcal{M}_r(e)$) is not accurate enough to perform a consistent analysis. We would have to define several sets, one for each function in the second case, and one for each "step" of the execution in the third case. Although it is conceivable, the impact on scalability would become an issue.

6.1.2.3 Scalability

The main argument for scalability is dynamic instrumentation, as mentioned in Section 4.2.1. However, we need to keep the instrumentation as lightweight as possible, to limit the overhead at execution due to the dynamic inspection. This overhead is highly dependent on the size of the sets $\mathcal{M}_p(e)$ and $\mathcal{M}_r(e)$, so they need to be wisely chosen. We propose in Chapter 8.2 definitions of these sets for a `x86-64` architecture and `System V AMD64 ABI` calling convention leading to a reasonable overhead (less than $\times 10$).

6.1.3 Arity over one execution

We propose in this section an approach over one execution e of a binary $\mathcal{B}$, in order to retrieve $\#_p f_e$ and $\#_r f_e$ for every $f \in e\lfloor f$. Then, in Section 6.1.5, we answer Problem 1 by a generalization to several executions.

6.1.3.1 Two steps analysis

Our approach is composed of two steps. First, we perform an instrumented execution e of the binary under analysis B. During this step, some events are logged (see Section 6.1.3.3) corresponding to what we need in order to solve Problem 1. Then, from the log obtained, we reconstruct some information relative to the execution - see Section 6.1.3.4.

6.1.3.2 Event counter

Each event we will log (see next section) will be indexed with a positive integer, to uniquely identify it and to order events that occur during the execution e. In particular, the same event can lead to several entries in the log ; in this case, every entry will correspond to the same value of the event counter, which makes explicit the fact that these entries are related to the same event. Index values start from 1, and are incremented by 1 each time a new event is logged. We denote by ec the event counter.

6.1.3.3 Inspection

We present in this section the inspections we need to perform during the execution e of the binary B in order to address Problem 1. There are two kinds of events we need to keep track of: function calls and returns, and location accesses.

Function calls and returns To be able to assign a parameter to a function, we need to keep track of a consistent stack of calls, relatively to the execution. To do so, we need to log each *call* and *return* (according to the definitions given in Section 5.2.1.1).

Event 1. *Function call or return:*

- *ec: the event counter,*
- *f: the function identifier,*
- *io: an indicator to know if f is being called (*`CALL`*) or is returning (*`RET`*).*

From these events, we show in Section 6.1.3.4 how to reconstruct a consistent stack of calls.

Location accesses We also log, during the execution e, every access to a memory location of $\mathcal{M}_p(e)$ and $\mathcal{M}_r(e)$, *i.e.*, potential locations for a parameter and/or return value.

Event 2. *Location access:*

- *ec: the event counter,*

- *a: the memory location being accessed (either a register or a memory address),*

- *io: an indicator to keep track of the nature of the access (*`W` *for write and* `R` *for read).*

With the knowledge of the stack of calls, we can deduce, for an access event ec, which function f is performing it by retrieving the last *call* event, anterior to ec, and that has no *return* corresponding event before ec. This is explained in the next section with more details.

6.1.3.4 Reconstruction

From the logged events described in Section 6.1.3.3, we reconstruct the stack of calls and the location accesses as follows.

Stack of calls We construct a function $StackOfCalls$, such that at any event counter ec of the execution e, $StackOfCalls(ec)$ returns an ordered list of events corresponding to calls that have not returned at event counter ec yet. We give in Algorithm 2 a naive algorithm to implement this function from the logged events, but the actual implementation uses high-level data structures and optimizations to be efficient on several calls (see Section 8.2).

Location In the same way, for any function f, we are able to extract from the log file the list of events that correspond to each call of f. For instance, $AccessEvents(f)$ returns a list of access events that occurred during the execution of f for each call to f. The output is therefore a list of lists of events (one list of events per call to f). As for the stack of calls, we give in Algorithm 3 an algorithm to implement this, but our actual implementation is more efficient (see Section 8.2). In addition, we construct two other functions, $PreviousWrite$ and $PreviousRead$ which return, for a given event counter ec and a memory location m, the latest event e in the log file that occurred before ec and that performed a write (for $PreviousWrite$) or a read (for $PreviousRead$) of m. An (naive) implementation of $PreviousWrite$ is given in Algorithm 4.

```
Function StackOfCalls(log, ec)
    Input: log a list of events sorted by event counter
    Input: ec max event counter to consider in the construction of the stack
           of calls
    begin
        S ← Stack();
        foreach event e in log do
            if e.ec > ec then
                return S;
            end
            if e is an instance of Event 1 then
                if e.io == CALL then
                    S.push(e)
                else
                    e' ← S.pop();
                    while e'.f ≠ e.f and not S.empty() do
                        e' ← S.pop();
                    end
                end
            end
        end
        return ∅
    end
```

Algorithm 2: Extraction of the state of the stack of calls at a given event counter ec

6.1.4 Heuristics

From the reconstructions presented in the previous section, we propose heuristics to deduce parameters of functions according to the execution e that was inspected.

6.1.4.1 Read before written

According to Definition 14, a parameter of f during the execution e is a memory location $m \in \mathcal{M}_p(e)$ that is read before written. This means that among every location of $\mathcal{M}_p(e)$, we need to find out the ones read before written by f and the ones that are not during e. For an execution $\varepsilon \in e(f)$ of f, and for a memory

location $m \in \mathcal{M}_p(e)$, three scenarios can occur:

1. *unused*: m is never accessed during ε,
2. *read before written*: m is read by an instruction of f before written during ε,
3. *write before read*: m is (over)written by an instruction of f before read during ε.

The intuition we use here is the following: the second scenario (*read before written*) leads directly to the conclusion that m is a parameter of f. However, scenarios of type 1 or 3 are *clues* for m not to be a parameter of f, but it is not enough to conclude: there might exist an execution ε' of f where m is read before written.

Heuristic 1. *Inside a function, one read before write on a memory location $m \in \mathcal{M}_p(e)$ means that m is a parameter of f, but a write before read or an unused one does not allow to conclude.*

As we cannot conclude on m in the case of *unused* and *write before read* only, and to we assume that m is not a parameter of f if for any $\varepsilon \in e(f)$, m is never read before written. However, in another execution e' of the same binary under analysis, there might exist such an execution ε of f. Algorithm 5 illustrates Heuristic 1 in pseudo-code. Regarding return values, we use the exact same argument and conclude that a *read before written* after a return implies a return value from the last function that returned.

6.1.4.2 Ascendant propagation

The intuition for this heuristic is that we need to consider parameters that are not actually used by the function f, but transmitted to another function g. Consider this simple example:

```
int f(int a) {
    int b = g(a);
    if (b % 2 == 0)
        return b / 2;
    else
        return 3*b + 1;
}
```

In this case, the value of a is never used by f, and if it is already at the correct memory location for the call to g, it is not read nor written, and thus would not be detected as a parameter of f. To answer this issue, we propose the *ascendant propagation* heuristic:

Heuristic 2. *A parameter detected for f_n and last written by f_0 is also a parameter of any f_i between f_0 and f_n in the stack of calls at the time f_n is called.*

6.1.4.3 Descendant propagation

The exact same argument stands for return values, illustrated by the following example:

```
int f(void) {
    return g();
}
```

Here again, the value returned by g will not be used by f, and because it is already at the correct memory location for return values, there is no need for f to access it. This leads to the *descendant propagation* heuristic:

Heuristic 3. *A return value detected for f_n and last written by f_0 is also a return value of any f_i between f_n and f_0 in the stack of calls at the time f_n returns.*

6.1.5 Generalization to answer Problem 1

The last two sections presented an approach and heuristics to retrieve $in_e(f)$ and $out_e(f)$ for any $f \in e\lfloor f$. To answer Problem 1, we need to discuss: first, how to deduce $in(f)$ and $out(f)$ from several $in_e(f)$ and $out_e(f)$; and second, the coverage of $\mathcal{F}(\mathcal{E}(\mathcal{B}))$.

6.1.5.1 Arity of f from a few executions

From Definition 16, we recall that

$$in(f) = \bigcup_{e \in \mathcal{E}(\mathcal{B})} in_e(f)$$

From a practical point of view, it is not realistic to perform every possible execution $e \in \mathcal{E}(\mathcal{B})$. We propose a heuristic to deduce $in(f)$ from $in_e(f)$ in practice.

Heuristic 4. *$in(f)$ can be computed from a small number of different executions $e \in \mathcal{E}(\mathcal{B})$ such that $f \in \mathcal{F}(e)$. This number can possibly be one.*

6.1.5.2 Coverage

Problem 1 is about finding the arity of every function $f \in \mathcal{F}(\mathcal{E}(\mathcal{B}))$. However, given a set of executions $\{e_1, e_2, ..., e_n\} \in \mathcal{E}(\mathcal{B})$, we can only deduce $\#_p f$ and $\#_r f$ for $f \in \bigcup \mathcal{F}(e_i)$. We address this question in our experiments (see Chapter 9.2), but in practice this is one limitation of our approach based on dynamic analysis: to analyze a given function f, we must observe at least one execution of f, *i.e.* we must forge an input of the binary $\mathcal{B}$ such that the execution leads to f.

6.1.6 Discussion

6.1.6.1 Validation

The method we propose is based on heuristics. To validate our approach, we propose to test it on a selection of open-source programs. Open-sourceness provides source-level information access that can be compared with the results of our analysis. These experiments and the results are detailed in Chapter 9.

6.1.6.2 Limitations

The main limitation of this heuristic-based approach is inherited from dynamic instrumentation: we cannot detect parameters that are not used during the executions of f we observe in e. We could suppose their existence, in some cases, using additional heuristics based on the calling convention ordering of parameters (see Section 8.2). However, doing this kind of specific deductions lead to a loss of generality (as it supposes that the calling convention specifies an order of parameters) and to more false positives.

6.2 Types

This section proposes an approach to retrieve types of parameters, based on the results of the arity detection. The general problem of this section is the following:

Problem 2. *Given a binary program $\mathcal{B}$ and a function $f \in \mathcal{F}(\mathcal{E}(\mathcal{B}))$ such that $in(f)$ and $out(f)$ are known, for any $i \in [\![1, \#_p f\ ]\!]$ and $j \in [\![1, \#_r f]\!]$, we want to retrieve $type(f_p[i])$ and $type(f_r[j])$.*

In Section 6.2.1, we present the types that we choose to consider. Section 6.2.2 refines Problem 2 according to our type set. We discuss criteria relatively to the problem in Section 6.2.3. Section 6.2.4 presents our approach to analyze one execution. We present our heuristics to answer the refined problem in Section 6.2.5, extends our answer to several executions in Section 6.2.6. Finally, we open a discussion in Section 6.2.7.

6.2.1 Set of types

We propose to focus the type detection on what we consider as the main semantic distinction in the use of data: addressing. One of the reasons why we consider this is because bad addressing is an important source of software crashes. The set of types we propose only aims to distinguish addresses, denoted by `ADDR`, from other kind of data that we denote by `NUM`. The following sections define explicitly the sets that will be used to address Problem 2 in our case.

6.2.1.1 Addresses

We define the type `ADDR` as follows:

$$\texttt{ADDR} = \{\texttt{LOAD}_1, \texttt{STORE}_\texttt{r}\}$$

where: `LOAD` is a generic operation taking two operands `l` and `r` such as `LOAD l, r` loads the content of the memory cell pointed by `l` into the memory location `r`; and `STORE` is a generic operation taking two operands `l` and `r` such that `STORE l, r` stores the value `l` (or the content of the memory location `l` to the memory cell pointed by `r`.

In other words, any value that is used as a left operand of `LOAD` *or as a right operand of* `STORE` *is an address.*

6.2.1.2 Numeric values

Our focus is on detecting addresses, so we do not need any other type. We say that a parameter that is not an address is a numeric value, denoted by `NUM`. To be compliant with definitions we proposed in Section 5.2.5.3, we could define another

set of operations named NUM, containing every operation except for $\texttt{LOAD}_1$ and $\texttt{STORE}_\texttt{r}$. Instead, we choose to redefine the set we denoted by UNDEF in Section 5.2.5.3: every parameter that is *undefined*, *i.e.*, the observed execution e does not include an operation of $\mathcal{O}(e)$, is considered as numeric value.

In other words, we rename UNDEF *into* NUM.

Another consequence of the unique type we are interested in is that we cannot have INCONS results. We recall that INCONS means that discriminating operations of different types were applied to the same parameter: because we only have one type (*i.e.*, one set of discriminating operations), this cannot happen.

6.2.1.3 Set of types and set of operations

In conclusion, we have only one type: $\mathcal{T} = \{\texttt{ADDR}\}$, and the set of operations we propose is composed of two operations: $\mathcal{O} = \{\texttt{LOAD}_1, \texttt{STORE}_\texttt{r}\}$. By extension, we consider NUM (*i.e.*, what is denoted by UNDEF in our definitions) as a second type, but not related to any specific operation.

6.2.2 Refinement of the problem

From the set of types we consider, we refine Problem 2:

Problem 3. *Given a binary program $\mathcal{B}$ and a function $f \in \mathcal{F}(\mathcal{E}(\mathcal{B}))$ such that $in(f)$ (resp. $out(f)$) is known, for any $i \in [\![1, \#_p f]\!]$ (resp. $j \in [\![1, \#_r f]\!]$), we want to determine if $f[i]$ (resp. $f[j]$) is of type* ADDR*.*

6.2.3 Criteria

In this section we discuss the implications of the three criteria of our approach relatively to type analysis.

6.2.3.1 Accuracy

Let us first define what we call *false positive* and *first negative* in the case of typing.

Definition 45. *A false positive is a parameter or a return value $p \in in(f) \cup out(f)$ of a function $f \in \overline{\mathcal{F}(\mathcal{E}(\mathcal{B}))}$ detected as an* ADDR *whereas $type(p) =$* NUM*.*

Definition 46. *A false negative is a parameter or a return value* $p \in in(f) \cup out(f)$ *of a function* $f \in \overline{\mathcal{F}(\mathcal{E}(\mathcal{B}))}$ *such that* $type(p) =$ `ADDR` *which is not detected as an* `ADDR`*.*

To be compliant with the target of accuracy, we try to avoid both false positives and false negatives.

6.2.3.2 Scalability

The type analysis is based on the arity results. This induces that only relevant memory locations of $\mathcal{M}_p(e)$ and $\mathcal{M}_r(e)$ are to be watched. Less memory locations to watch means less instrumentation. However, we need to perform analysis of every `LOAD` and `STORE`. We limit as much as possible the actions to perform when handling these instructions. Experimental results (see Chapter 9) shows that we keep a scalable approach doing this.

6.2.3.3 Universality

Because this step comes after the detection of arity, it inherits the limitations due to this step (unknown calling convention, none or multiple calling conventions). However, we do not introduce in this step of type detection new assumptions that would lead to a restriction of the set of binaries we can target.

6.2.4 Type over one execution

The approach we propose for type detection is very similar, in its design to the approach used for arity detection (see Section 6.1.3). Once again, we present in this section an analysis of one execution e of the binary under analysis $\mathcal{B}$. It is also in two steps, one for inspection - see Section 6.2.4.1, and one for reconstruction - see Section 6.2.4.2. Our approach is based on events to be logged, that differ in a few points from the events presented in Section 6.1.3.3. We keep the same definition of event counter that we proposed in Section 6.1.3.2, *i.e.* an integer value that is incremented each time an event is logged, with an exception that we present in the next section.

6.2.4.1 Inspection

Calls and returns In opposition to the arity problem, we do not need here to keep a consistent stack of calls. However, we need to instrument every call of

functions that take more than zero parameter, and every return of those that return more than zero value. More precisely, for every $f \in \mathcal{F}(\mathcal{E}(\mathcal{B}))$,

- if $\#_p f > 0$, we instrument every call to f,
- if $\#_r f > 0$, we instrument every return from f.

At each call (resp. return) instrumented, for every $m \in in(f)$ (resp. $m \in out(f)$), we log an event which includes the concrete value stored at m.

Event 3. *Value of parameter or return value:*

- *ec: the event counter,*
- *f: the function being called or returning,*
- *io: an indicator to know if f is being called (*`CALL`*) or is returning (*`RET`*),*
- *i: the position of the parameter (resp. return value) being logged (i.e., its position in the ordered set $in(f)$ (resp. $out(f)$),*
- *v: the concrete value of the parameter (resp. return value), i.e. the value stored at the memory location corresponding to the parameter $f[i]$.*

Incrementation of the event counter In Section 6.1.3.2, we defined the event counter as an integer that is incremented each time an event is logged. However, in this case, we want to be able to identify parameters (or return values) relative to the same call. Therefore, even if we log several events for several parameters, we only increment *ec* when a new call or return is instrumented, or one of the events presented in the next section occur. This means that three concrete values of three different parameters of a function f corresponding to the same call will be logged with the same *ec*.

`LOAD` **and** `STORE` To detect addresses, we also need to instrument `LOAD` and `STORE` instructions. In other words, we log the operands of every instruction that uses it for addressing. The logged events have the following format:

Event 4. *Memory access:*

- *ec: the event counter,*
- *m: the concrete value being used as an address,*
- *io: an indicator to know if the location m was read (*`R`*) or written (*`W`*).*

6.2.4.2 Reconstruction

To conclude on the type of a parameter, we need two kind of offline analysis based on the log of events produced by the inspection as described in the previous section.

Concrete values First, for any function f, we need to get the concrete values of the i^{th} parameter (resp. return value) of f during e, for any $i \in [\![1, \#_p f]\!]$ (resp. $i \in [\![1, \#_r f]\!]$). Algorithm 6 presents a possible implementation of this for parameter values. However, for the same reasons we detailed for arity, the actual implementation is more efficient (but also less readable). A very similar implementation applies to the detection of return values.

Memory operands Second, we need to be able to say, for any concrete value v, if it has been used as a memory operand during the execution e. Algorithm 7 parses the log file and return `true` if an memory access event was log with v as a memory operand, and `false` otherwise.

6.2.5 Heuristics

In this section, we propose heuristics to conclude, from an execution e, on $type_e(p)$ for any $f \in \mathcal{F}(e)$ and any $p \in in_e(f) \cup out_e(f)$. In Section 6.2.6, we generalize this to conclude on $type(p)$ independently of a given execution e.

6.2.5.1 Value-based detection

First, we assume that integer values and address values have a very small probability to collide in a normal execution. Typically, integers will have small values whereas addresses will be around 2^{64} on 64bit architectures. In other words, we assume that the set of values for integers is disjoint from the set of values for addresses. This leads to the first heuristic:

Heuristic 5. *Once a value is typed, every further occurrence of the same value has the same type.*

This means that we generalize the definition of $type_e$ to values, and that once a value v is typed as an address, any further occurrence of it will be considered as an address as well. In opposition to parameters, and although the type of a value is detected by the use of v in a particular execution e, it has no reason to be dependent on e. We denote by $type(v)$ its type.

Discussion This is an over-approximation. Indeed, an integer that would have the same value v as an address would be considered as an address. We discuss in Section 6.2.7.1 how we deal with this regarding accuracy (and soundness).

6.2.5.2 Addressing

Second, we propose a heuristic to detect some addresses from the logged events, assuming that a value v is an address as soon as it is used as it.

Heuristic 6. *A value used as a memory operand is an address.*

In other words, every value v such that $IsMemOp(log, v)$ is true is considered as an address. This, in addition to Heuristic 5, means that we can detect v as an address the moment it is used.

6.2.5.3 Address parameters

We use a strong heuristic to infer data flow from the observations of the logged events.

Heuristic 7. *For any function $f \in \mathcal{F}(e)$, an execution $\varepsilon \in e\lfloor f$, and for any $p \in in_\varepsilon(f) \cup out_\varepsilon(f)$ getting at some point the concrete value v during ε, if $type(v) = \texttt{ADDR}$, then $type_\varepsilon(p) = \texttt{ADDR}$.*

This heuristic means that if a function f takes a value v as a parameter p, and if this same value v is used later or was used before as a memory operand, , then we assume a data flow (as defined in Definition 27) between p and the use of v. Assuming this data flow allows to conclude on the type of p from the type of v.

6.2.5.4 Once an address, always an address

Finally from several executions $\varepsilon_1, \varepsilon_2, ...\varepsilon_n \in e\lfloor f$ of a function $f \in \mathcal{F}(e)$, we deduce $type_e(p)$ for every parameter $p \in in_e(f) \cup out_e(f)$ with the following heuristic.

Heuristic 8. *If a parameter p of f is detected as an address in one execution $\varepsilon \in e\lfloor f$, then we deduce that its type in e is always* ADDR*.*

$$type_e(p) = \begin{cases} \texttt{ADDR} \text{ if } \exists \varepsilon \in e\lfloor f, type_\varepsilon(p) = \texttt{ADDR} \\ \texttt{NUM} \text{ otherwise} \end{cases}$$

6.2.5.5 Algorithm

These heuristics lead to a simple algorithm to detect parameters that are addresses according to the logged event of one execution - see Algorithm 8.

6.2.6 General answer to Problem 3

To generalize from $type_e(p)$ to $type(p)$, we use the same arguments as presented in Section 6.1.5 regarding arity. From Definition 26, we recall that for a given parameter p,

$$type(p) = \bigcap_{e \in \mathcal{E}(\mathcal{B})} type_e(p)$$

In our case, using Heuristic 8, it means that if there is any execution $\varepsilon \in \mathcal{E}(\mathcal{B})\lfloor f$ such that $type_\varepsilon(p) = \texttt{ADDR}$, we deduce that $type(p) = \texttt{ADDR}$. This allows to conclude that a parameter p is an address without exploring every possible execution e of $\mathcal{E}(\mathcal{B})$.

6.2.7 Discussion

We discuss in this section some particular points regarding our heuristics. As for arity, a heuristic-based approach requires experiments to be validated. We present our implementation and results in Sections 8.3 and 9.3.

6.2.7.1 Accuracy

As mentioned in Section 6.2.3.1, a false positive would be to detect a parameter p as an address (`ADDR`) whereas it is in reality a numeric value (`NUM`). With our approach, it can occur in practice with large integer values. In particular, there can be some "unfortunate" collisions between big integers and address values. Heuristic 8 would deduce a dataflow, in such a case, that does not exist in practice. This problem can occur, in particular, if we deal with random generators which are good candidates to output various integers with high values that could collide with addresses. This point could be addressed specifically, with a more refined dataflow tracking (using other heuristics), but it would be to the cost of scalability. In practice, we show in Chapter 9 that the accuracy of type detection is very acceptable and that the rate of false positives is lower than the rate of false negatives.

6.2.7.2 Variable typing

The general answer to Problem 2 hides another assumption to deduce the general type of a parameter from several executions of a given function: we assume that a parameter p is always of the same type (this is the meaning of Heuristic 8). However, we could have parameters whose types depend on the context of the call. In such paradigms, our approach will lead to conclude with `INCONS` for a lot of parameters (if we would consider more types than `ADDR` only). However, such behaviors are not common in compiled computer programs.

```
Function AccessEvents(log, f)
    Input: log a list of events sorted by event counter
    Input: f the function from which we want to extract events
    begin
        access_events ← List();
        curr_events ← List();
        curr_call ← None;
        foreach event e in log do
            if e is not an instance of Event 2 then
                continue;
            end
            last_call ← StackOfCalls(log, e.ec).top();
            if last_call.f = f then
                /* If a new call to f occurred, we need to store
                   the  list of events corresponding to the
                   previous call                                  */
                if last_call.ec ≠ curr_call.ec then
                    access_events.add(curr_events);
                    curr_events ← List();
                    curr_call ← last_call;
                end
                curr_events.add(e);
            end
        end
        return access_events;
    end
```

Algorithm 3: Extraction of the memory location events relative to a given function f

```
Function PreviousWrite(log, m, ec)
    Input: log a list of events sorted by event counter
    Input: m the memory location of interest
    Input: ec the event counter from which we want to retrieve the previous
           write
    begin
        prev_event ← None;
        foreach event e in log do
            if e is not an instance of Event 2 then
                continue;
            end
            if e.ec >= ec then
                return prev_event;
            else if e.a = m and e.io = W then
                prev_event ← e;
            end
        end
        return prev_event;
    end
```

Algorithm 4: Retrieving the latest write access to m that occurred before ec

```
Function ReadBeforeWritten(log, f)
    Input: log a list of events sorted by event counter
    Input: m the memory location of interest
    Input: ec the event counter from which we want to retrieve the previous
           write
    begin
        read_before_written ← Set();
        foreach event e in log do
            if e is not an instance of Event 2 then
                continue;
            end
            prev ← PreviousWrite(log, e.a, e.ec);
            curr_f ← StackOfCalls(log, e.ec).top();
            if StackOfCalls(log, prev.ec).top() ≠ curr_f then
                /* read_before_written is a set, so add will be
                   effective only if e.a is not already in the
                   set                                          */
                read_before_written.add(e.a);
            end
        end
        return read_before_written;
    end
```

Algorithm 5: Procedure to detect *read before written* events and conclude on the existence of parameters

```
Function ConcreteValuesP(log, f, i)
    Input: log a list of events sorted by event counter
    Input: f the function considered
    Input: i the position of the parameter of f of which we want concrete
           values
    begin
        vals ← List();
        foreach event e in log do
            if e is an instance of Event 4 then
                continue;
            end
            if e.f = f and e.i = i and e.io = CALL then
                vals.add(e.v);
            end
        end
        return vals;
    end
```

Algorithm 6: Get concrete values taken as parameter during an execution e for a given function f

```
Function IsMemOp(log, v)
    Input: log a list of events sorted by event counter
    Input: v the value to be typed
    begin
        foreach event e in log do
            if e is an instance of Event 3 then
                continue;
            end
            if e.v = e then
                return true;
            end
        end
        return false;
    end
```

Algorithm 7: Determining if a given value v is a memory operand

```
Function Type(log, f, i)
    Input: log a list of events sorted by event counter
    Input: f the function under analysis
    Input: i the position of the parameter of f to infer
    begin
        foreach value v in ConcreteValuesP(log, f, i) do
            if IsMemOp(log, v) then
                return ADDR;
            end
        end
        return NUM;
    end
```

Algorithm 8: Address parameter detection

Chapter 7

Inference of address flow

In this chapter, we focus on address flow between functions. Based on the structures of functions we infer in Chapter 6, we are interested in the production and the use of ADDR values. Section 7.1 introduces an approach to retrieve couples of functions relative to ADDR parameters, and Section 7.2 focuses on detecting memory allocators.

7.1 Coupling

In this section, we present an approach to retrieve the notion of *coupling* we proposed in Section 5.3.4, which aims to capture a behavioral pattern at function level. We first present the intuitions of this pattern in Section 7.1.1. In Section 7.1.2, we define the problem we address. Section 7.1.3 discusses this problem relatively to our criteria. The approach and our heuristics to target one single execution are presented in Sections 7.1.4 and 7.1.5. In Section 7.1.6, we extend this to answer Problem 4. Finally, we discuss our approach in Section 7.1.7.

7.1.1 Intuitions

Our work focuses on functions, and one particularity of functions is that they usually output values and take parameters. A behavioral analysis includes to determine how return values are used later in the execution, and where the parameters of functions come from. The notion of coupling aims to describe a dynamic pattern in return

value reuse. More specifically, it targets functions f and g such that a parameter of g is "*often*" a value output by f. Details about "often" and what we mean by this were given in Section 5.3.4, but the intuition here is to find interactions between functions. This is interesting, from a reverse-engineering process point of view, because it gives a good idea of both data flow and control flow with a single representation that is simple to get. Saying that f an g are coupled means that f returns values that are given to g as parameters, which allows to deduce that f is often call before g and that there is a data flow from f to g.

7.1.2 Problem

We define in this section the problem we address. First, we refine the notion of coupling, based on the notion of type we introduced in Section 6.2.1.

7.1.2.1 Strong coupling

The set of types we consider in this work is composed of two types: `NUM` and `ADDR`. We propose a restriction to the definition of coupling to focus on addresses.

Definition 47. *For $(f, g) \in \mathcal{F}(\mathcal{E}(\mathcal{B}))^2$ and $(i, j) \in [\![1, \#_r f]\!] \times [\![1, \#_p g]\!]$, we say that $f_r[i]$ and $g_p[j]$ are strongly ρ-coupled for a given $\rho \in [0, 1]$ if $f_r[i]$ and $g_p[j]$ are typely ρ-coupled and $type(f) = type(g) =$ `ADDR`.*

7.1.2.2 Definition of the problem

In this work, we focus on the detection of functions that are strongly coupled, according to a coupling rate ρ. We do not target coupling of functions that does not include address parameters. The problem we address is, thus, to retrieve a subset of the functions that are ρ-coupled according to Definition 31.

Problem 4. *For a given couple rate $\rho \in [0, 1]$ and a binary $\mathcal{B}$, we want to retrieve every $(f, g) \in \mathcal{F}(\mathcal{E}(\mathcal{B}))^2$ and $(i, j) \in [\![1, \#_r f]\!] \times [\![1, \#_p g]\!]$, such that $f[i]$ and $g[j]$ are strongly ρ-coupled.*

7.1.3 Criteria

7.1.3.1 Accuracy

Because coupling is a statistic notion, it is not relevant to define a notion of accuracy relatively to the global definition. We could say that a false positive would be a couple (f, g) detected whereas it does not exist, and a false negative would be a couple (f, g) not detected whereas it holds, but we would rather focus on false positives and negatives regarding the *def-use* chains. In this sense, we call false positive a value that is considered to be output by f and taken as a parameter by g whereas it is not.

Definition 48. *We call false positive a def-use chain between f and g detected but that does not exist.*

A false negative would be a value that is output by f and taken as a parameter by g but not detected.

Definition 49. *We call false negative a def-use chain between f and g not detected but that does exist.*

Here again, we aim to limit the number of both false positives and false negatives.

7.1.3.2 Scalability

As mentioned in more details in Section 7.1.4.1, the inspection we need to perform to address Problem 4 is a subset of the inspection relative to coupling. In addition, we propose several heuristics in our experiments to lighten the instrumentation - see Section 8.4. In conclusion, this analysis is less expensive, in overhead, than the type detection presented in Section 6.2.

7.1.4 Coupling over one execution

7.1.4.1 Inspection

To perform couple detection, we need an instrumentation which is close to the one presented in Section 6.2.4.1 for type recovery. In fact, the inspection we propose here can be seen as a subpart of it. This means in particular that we could perform couple detection from the same log file. This will be discussed in Section 7.1.7. We

present here the instrumentation required for coupling, *i.e.* the minimum events to log.

Calls and returns As in the inspection for types, we need concrete values of parameters and return values of functions. To do so, we instrument calls and returns: each time a function f is called (resp. returns), a handler is called to log concrete values of parameters (resp. return values) $p \in in(f)$ (resp. $p \in out(f)$). Therefore, this analysis is based on the results of arity detection. The logged events have the same format as defined in Event 3.

Addresses involved However, this analysis differs from the one relative to type detection as it uses the results of type inference to lighten the instrumentation. Because we are only interested in detecting **strong** coupling, *i.e.* coupling involving addresses, we only need to instrument, during the step, functions that have an address in their undertyped prototype, and to log only parameters that are of type `ADDR`. Table 7.1 shows examples of functions that we do or do not instrument according to their `C`-like prototypes. The first symbol (✓ or ✗) corresponds to the call instrumentation, and the second symbol to the return instrumentation.

in \ out	`VOID f(...`	`NUM f(...`	`ADDR f(...`
`VOID)`	✗ / ✗	✗ / ✗	✗ / ✓
`NUM)`	✗ / ✗	✗ / ✗	✗ / ✓
`ADDR)`	✓ / ✗	✓ / ✗	✓ / ✓
`NUM, NUM, ADDR)`	✓ / ✗	✓ / ✗	✓ / ✓

Table 7.1: Instrumentation of calls/returns of functions according to the type of their parameters/return values

Sampling In addition to only instrumenting a subset of functions of $\mathcal{F}(\mathcal{E}(\mathcal{B}))$, we do not log every concrete value. Indeed, we propose in Section 7.1.5 to detect coupling on a finite (small) sample of concrete values. We only instrument the first N calls to each function, and get the N first concrete values of $f[i]$ for any $i \in [\![1, \#f\ ]\!]$ such that $type(f[i]) =$ `ADDR`. N is a parameter of the instrumentation to be specified (see Section 9.4 for details on the influence of N). Note that

despite N, every return value is logged. The reason we do this is explained in Section 7.1.5.2.

7.1.4.2 Reconstruction

To retrieve couples from the logged events, we require two reconstructions: the set of concrete values of parameters and return values, and a function to compute the intersection of two sets.

Parameter and return values This is the same reconstruction as presented in Section 6.2.4.2: we need two functions $ConcreteValuesP$ and $ConcreteValuesR$ to get the concrete values of parameters and return values of a given function and a given position of the parameter. The implementation is the same as presented in Algorithm 6.

Intersection of sets We also need a function $Intersect$ that takes two sets of values as parameters, and returns the intersection of them, *i.e.* a set of values that are in both input sets. The implementation is straight-forward, so the algorithm is not presented here.

7.1.5 Heuristics

We present here two heuristics we use to retrieve strong couples from the inspection of an execution e as explained in the previous section.

7.1.5.1 Collision implies dataflow

Our first heuristic is similar to Heuristic 7, and is relative to data fow. The definition of coupling is based on data flows beteen return values and parameters. However, computing data flow dynamically is expensive. On the other hand, the address space is large (2^{64} elements in 64bit architectures), so an unfortunate collision is very unlikely. That is the reason why we assume a data flow between an address return value and an address parameter as long as they have the same concrete value.

Heuristic 9. *For a given execution e, given two functions f and g in $\mathcal{F}(\mathcal{E}(\mathcal{B}))$, a return value $r \in out_e(f)$ and a parameter $g \in in_e(g)$ such that $type(p) = type(r) =$ `ADDR`, if r takes the concrete address value a and if p takes the same*

address value a further in e, we assume a data flow from r to p according to Definition 27.

In other words, two occurrences of the address value implies a data flow.

7.1.5.2 Sampling

Definition 31 is not symmetric. In particular, the parameter ρ is the proportion of *parameter values* that comes from the return values of another function. We propose to compute this proportion on a sample of values instead of all values observed during e.

Heuristic 10. *For r a return value, and p a parameter, we can compute the rate coupling ρ of the couple (r, p) over an execution e on a sample of concrete values of p.*

However, due to the asymmetry of the definition of coupling, although we can sample parameter values, we still need to get every concrete return value. Otherwise, we could miss some *def-use* chains between r and p because the sample of concrete values of r would not contain a given concrete value v of p.

7.1.5.3 Algorithm

From these two heuristics, we propose in Algorithm 9 a possible implementation to determine if two functions f and g in $\mathcal{F}(e)$ are strongly ρ-coupled relatively to an execution e, and according to a given coupling rate ρ.

7.1.6 Generalisation to answer Problem 4

Similarily to what we did for arity and type, we assume that, from one execution e, we can conclude on coupling on any execution $e \in \mathcal{F}(\mathcal{E}(\mathcal{B}))$. If we have several executions $e_1, e_2, ...e_n$, and for each a list of logged events log_i, we generalize Algorithm 9 to perform the computation on more concrete values. We recall that for each execution, we keep at most N concrete values for each parameter p. With n executions, we can thus perform the computation of coupling on largest data set (at most $n * N$). What is interesting is that these values are obtained from different executions ; we assume that they capture more possible cases than if we get $n * N$ values from one execution (supposing that there are enough executions of g).

7.1.7 Discussion

7.1.7.1 Rupture of dataflow

In our approach, Heuristic 9 induces that we do not consider ruptures in dataflow: from a collision of values we deduce a def-use chain (based on this dataflow). This can seem drastic at first: if there are many instructions executed between the *def* and the `use`, there are many chances that there is no dataflow anymore between the two (because of overriding at some point). However, because we are tracking addresses, this is not a problem: twice the same address at two different points of the program point at the same memory cell, and this does not depend on the concrete dataflow of the address between these two points. The content may have been overwritten, but both accessors can read/write in the same cell. In addition, if we know who allocates memory (see Section 7.2 for allocator retrieving): we could consider a free of a given cell as a rupture of data flow.

7.1.7.2 Inspection

As mentionned before, we propose an instrumentation to perform coupling detection over one execution with as less overhead as possible. However, it would be possible to perform both type and coupling detection from the same trace (with the same instrumentation of the execution). For clarity purpose, we decide to keep a clear separation between these two steps and each one is performed with a specific instrumentation.

7.2 Memory allocator

7.2.1 Motivations

Many harmful program defects and security vulnerabilities are due to memory errors occurring in the heap (see [QLZ05] and [CGMN12] for instance). For this reason, memory accesses are subject to many researches ([RB08], [BR04], [VCKL05]). Both static and dynamic existing techniques allowing to detect such heap related issues often heavily rely on some *a priori* knowledge of memory allocation and liberation functions[1]. For instance, `Valgrind` [NS07] requires to know the allocator to perform a memory leak analysis. However, numerous real-life application now

[1] like the classical `malloc()` and `free()` in `C` programs

embed their own allocators, and retrieving the corresponding function pair could be quite challenging, in particular when the source code is not (fully) available. In this section, we propose an approach to retrieve allocators from a binary program. First, we state the problem we address in Section 7.2.2. Section 7.2.3 gives some precisions about our approach relatively to the criteria we chose in Section 4.1.3. We present our instrumentation and reconstruction algorithms over one particular execution in Section 7.2.4. In Section 7.2.5 and Section 7.2.6, we propose heuristics and algorithms to retrieve respectively `ALLOC` and `FREE` from the instrumented execution.

7.2.2 Problem

Problem 5. *From an execution e of a binary program $\mathcal{B}$, retrieve the couple (`ALLOC`, `FREE`) corresponding to the most frequently used allocator during e (according to Definition 42).*

7.2.3 Criteria

In this section, we discuss Problem 5 relatively to the three criteria we focus on.

7.2.3.1 Accuracy

Our approach to answer Problem 5 outputs a couple (`ALLOC`, `FREE`) detected as the allocator of an execution e. For this problem, it is not really relevant to define false positives and false negatives, and so soundness and completeness, as there should be exactly one *most frequently used allocator* during an execution e (or none if the program never allocates memory). However, `ALLOC` could be used in further analysis, for example to follow the use of memory blocks after allocation, or to track use-after-frees. Thus, we propose ways to test the result of the detection of (`ALLOC`, `FREE`) - see Section 9.5, to conclude on the consistency of this couple as an allocator. If the test fails, then the output of our final detection would be `None` (or equivalent). In this context, we can define a false positive as a concrete output couple (`ALLOC`, `FREE`) that does not correspond to an allocator. A false negative would be the output of `None` whereas the binary under analysis performs allocations during e. To be sound-oriented, we want our approach to ouput only correct allocators.

7.2.3.2 Universality

The definition of allocators we proposed in Section 5.4 aims to be as generic as possible, and to capture the largest set of different implementations of memory allocators (see Section 9.5.2). For instance, we do not use strong asumptions on the prototype of `ALLOC` and `FREE` to drive our detection ; although such asumptions could be used to validate efficiently the consistency of our results - see Section 9.5.1. However, our inspection may exclude some particular cases of allocators by design. We discuss in Sections 7.2.5.3 and 7.2.6.2 limitations of our approach, respectively for `ALLOC` and `FREE`.

7.2.3.3 Scalability

The instrumentation we propose is a subset of the instrumentation performed for type detection - see Section 6.2.4.1. The scability of allocator detection during execution is thus bounded by the scability of type detection. In fact, we use several arguments to reduce the inspection to functions that are potential candidates for `ALLOC` and `FREE` (see next section), and do not instrument every function of $\mathcal{F}(e)$. However, this approach has an expensive offline step (see Section 9.5), and could be a problem with very large traces of execution - see Section 9.5.

7.2.4 Allocator over one execution

In this section, we present the online (Section 7.2.4.1) and offline (Section 7.2.4.2) steps to retrieve the most frequently used allocator over one execution e. From these steps, we propose in Section 7.2.5 heuristics to answer Problem 5.

7.2.4.1 Inspection

The inspection we propose here is a subset of the inspection for types. Actually, we only need to instrument `CALL` and `RET` events, using the exact same format as presented in Event 3. By opposition to coupling instrumentation, we need every call and every return to be logged. However, we do not need to instrument every function, and among these functions we do not need to log every concrete value of parameters.

Strong couples First, we only instrument a function $f \in \mathcal{F}(e)$ if there exists a function g and two indexes i and j such that $f[i]$ and $g[j]$ or $g[j]$ and $f[i]$ are stronly

ρ-coupled, with $\rho = 0.9$ (this is a parameter of the instrumentation). The idea behind this is that `ALLOC` and `FREE` must be strongly coupled with a high coupling rate, so there is no need to instrument functions that are not part of couples.

Address parameters Second, as we target memory allocators, we are not interested in parameters and return values that are not of type `ADDR`. For every function that is instrumented (*i.e.*, every function coupled with at least one other function with $\rho > 0.9$), we only log concrete values of parameters and return values that are addresses. If a function has no parameter (or return value) of type `ADDR` but needs to be instrumented, we log an empty event to get an entry corresponding to the call (return): this is needed to keep a compliant stack of calls.

7.2.4.2 Reconstruction

To reconstruct data from the logged events, we reuse some algorithms we proposed in the previous chapters, and we need another algorithm.

Reuse From the previous analysis, we need the following points:

- $StackOfCalls$ (Algorithm 2) - get the concrete state of the stack of calls at event counter ec given in parameter.
- $ConcreteValuesR$ (Algorithm 6) - get the concrete values output by a given function as i^{th} return value,
- $type$ (Section 6.2) - for any parameter of any function of e, we assume that we did retrieve its type and that it is available through a function $type : f[i] \leftarrow type(f[i])$. The same stands for $type_r$ which gives the type of return values.

Number of callers In addition, we need to retrieve, for every function $f \in \mathcal{F}(e)$, the number of unique callers $c \in \mathcal{F}(e)$, *i.e.* the functions that, at some point of the execution e, called f. This is used in our heuristics to detect `ALLOC` and `FREE`, and in particular to check the criterion of diversity of callers that we present in Sections 7.2.5.1 and 7.2.6.1. We propose an implementation from the logged events corresponding to e in Algorithm 10.

7.2.5 Retrieve ALLOC

To retrieve the most frequently used allocator, we propose first to detect ALLOC. The knowledge of ALLOC will be used then to retrieve FREE with a second offline analysis - see Section 7.2.6.

7.2.5.1 Heuristics

We propose three heuristics to retrieve ALLOC relatively to ne execution.

Allocation and addresses According to Definition 42, an allocating function is a function which transfers an unallocated block $(p, s) \in \mathcal{B}(\overline{\rho\downarrow})$ to the allocated memory. For this newly allocated block to be usable by the program, this function must output a way to access this block. Our heuristics assume this is done by outputting a pointer to the block.

Heuristic 11. *The allocating function outputs as a return value the address p of the allocated block* (p, s)*.*

Diversity of callers Usually, allocators present a public interface (ALLOC, FREE) but use several internal functions, either to find a block to allocate (according to a given strategy) or to merge free blocks. These internal functions manipulate blocks as much as the public interface, however we are not interested in their detection. As they are internal, they cannot be called by a lot of different functions. We use this as a criterion to eliminate them from the possible candidates for (ALLOC, FREE).

Heuristic 12. *The less a function f is called by various other functions, the less it is likely that f is an allocating function.*

Production of new addresses Finally, our main heuristic is relative to the production of new addresses. Heuristic 11 emphasis the relation we consider between allocation and output of addresses. In facts, only allocators are, by definition, able to allocate memory. This means that every address output by a function is either the pointer to a new allocated block or the address corresponding to an already allocated memory. In our heuristic, we assume that an allocation mainly outputs new addresses, and other functions do not use a lot of addresses never seen before.

Heuristic 13. *The more a function f outputs new addresses (never seen before), the more it is likely that f is an allocating function.*

An implementation to compute the number of new addresses a function has produced during an execution is presented in Algorithm 11. Note that we assume we know the type of each parameter of functions (*i.e.*, we have already performed a type analysis of e).

7.2.5.2 General algorithm

From the three heuristics we proposed in the previous section, in Algorithm 12 we propose to compute a score for every function $f \in \mathcal{F}(e)$, and to output the best candidate for `ALLOC`. In this algorithm, we choose to set the criterion *diversity of callers* to 3: indeed, more than 3 callers means that it is not an internal function of the allocator only called by `ALLOC` and `FREE`

7.2.5.3 Discussion

Non unicity of `ALLOC` Our heuristics target one allocator (the one with the highest score, regarding our criteria), although there might be several ones in a given program. For example, one custom allocator can be used in the core of a given program whereas the standard libc allocator (`malloc`, `free`) would be used in libraries. Our approach focuses on the most frequently used (our score is computed in this way), but it could be adapted to detect allocators using other criteria. To avoid the detection of (`malloc`, `free`) in case a program embeds a custom allocator and uses a lot of library functions (these functions do use `malloc` and `free` and not the custom allocator), we can ignore, for example, any call from a library to a library for this analysis. This approach is proposed and implemented in Section 8.5, and allows to detect the custom allocator instead of the one used by libraries.

Wrappers The case of wrappers and layers of allocators is a little bit specific. Due to the *diversity of callers* criterion presented in Heuristic 12, a wrapper around an allocator will hide the real allocator from detection. For example, see Listing 7.1. In this example, `wrapper_alloc` is a wrapper around `malloc` which performs a safety check and returns the allocated block. In this scenario, `malloc` is never called directly (except through `wrapper_alloc`), so it will not be detected as an allocator (because of the *diversity of callers* criterion). On the other hand, `wrapper_alloc`

will likely be detected as ALLOC because, if we ignore malloc, it produces new addresses and probably has more different callers. This is a minor issue, as it is easy to retrieve malloc from wrapper_alloc - see Section 8.5.

```
void *safe_alloc(size_t size) {
    void *ptr = malloc(size);
    if (ptr == NULL)
        exit(1);
    return ptr;
}
```

Listing 7.1: Example of a wrapper around malloc in C

Based on prototypes Detection of allocators is based on the results of the prototype inference. In this approach, we assume that prototypes were infered correctly. Incorrect typing could lead to erroneous deduction regarding ALLOC (and thus FREE). For example, if a random generator, used many times during an execution, is inferred as outputting an address, then it will end up with a high score (many addresses produced and a sufficient variety of callers) and might be considered as an allocator.

```
Function StrongCouples(log, ρ)
    Input: log a list of events sorted by event counter
    Input: ρ the coupling rate
    begin
        couples ← List();
        foreach function g in F(e) do
            foreach parameter j in [[1, #_p g_e]] do
                if type_e(g[j]) ≠ ADDR then
                    continue;
                end
                p_vals ← ConcreteValsP(log, g, j).size();
                nb_tot ← p_vals.size();
                foreach function f in F(e) do
                    foreach i in [[1, #_r f_e]] do
                        if type_e(f[i]) ≠ ADDR then
                            continue;
                        end
                        r_vals ← ConcreteValsR(log, f, i).size();
                        nb_shared ← Intersect(p_vals, r_vals).size();
                        if nb_shared/nb_tot > ρ then
                            couples.add((f[i], g[j]));
                        end
                    end
                end
            end
        end
        return couples;
    end
```

Algorithm 9: Recovering strong ρ-couples from an execution e

```
Function NbCallers(log, f)
    Input: log a list of events sorted by event counter
    Input: f the function of interest, from which we cant to compute the
           number of callers
    begin
        callers ← Set();
        foreach event e in log do
            if e is an instance of Event 3 and e.f = f then
                /* Get the stack of calls just before f was
                   called, and take the top element            */
                caller ← StackOfCalls(log, e.ec − 1).top();
                /* callers is a set, so caller will only be added
                   if not already in the set                   */
                callers.add(caller);
            end
        end
        return callers.size();
    end
```

Algorithm 10: Compute the number of callers of a given function f during an execution e

```
Function NbNewAddr(log, f, i)
    Input: log a list of events sorted by event counter
    Input: f the function of interest, from which we cant to compute the
           number of callers
    Input: i the return value considered
    begin
        nb_new ← 0;
        addr_seen ← Set();
        foreach event e in log do
            if e is not an instance of Event 3 then
                continue;
            end
            if (e is CALL and type(e.f[e.i]) ≠ ADDR) or (e is RET and
              type_r(e.f[e.i]) ≠ ADDR) or e.v ∈ addr_seen then
                continue;
            end
            if e is RET and e.f = f and e.i = i then
                nb_new ← nb_new + 1;
            end
            addr_seen.add(e.v);
        end
        return nb_new;
    end
```

Algorithm 11: Compute the number of new addresses output by f as the i^{th} return value during the execution e

```
Function GetAlloc(log)
    Input: log a list of events sorted by event counter
    begin
        alloc ← None;
        retidx ← None;
        best_score ← 0;
        foreach function f in F(e) do
            if NbCallers(log, f) < 3 then
                continue;
            end
            foreach i ∈ ⟦1, #_r f⟧ do
                score ← NbNewAddr(log, f, i);
                if score > best_score then
                    best_score ← score;
                    alloc ← f;
                    retidx ← i;
                end
            end
        end
        return (alloc, retidx);
    end
```

Algorithm 12: Retrieve `ALLOC` from an execution e

7.2.6 Retrieve FREE

Once we have retrieved ALLOC = $(alloc, retidx)$, we can use this result to retrieve FREE. From the logged events, and with some heuristics, we can output the best candidate to be the FREE function that corresponds to ALLOC.

7.2.6.1 Heuristics

Freeing and addresses According to Definition 41, a freeing function is a function that transfers an allocated block (p, s) to the free memory. To do so, this function must have a way to known which block needs to be freed. In this heuristic, we assume that it is given as a parameter by passing a pointer.

Heuristic 14. *The freeing function takes as a parameter the address p of the block (p, s) to be freed.*

Diversity of callers We reuse here Heuristic 12, that we presented relatively to ALLOC, with the same argument. Allocating and freeing functions are susceptible to be called by any other function, so the diversity of their callers is a clue: a function that is called by a very small number of different functions is not a good candidate for FREE.

Last accessor The main criterion to retrieve FREE, in our approach, is based on the definition: FREE "moves" a block from the allocated memory to the free memory, which means that this block should not be accessed after its liberation. Consequently, FREE should be the last accessor of a memory block (before it is reallocated).

Heuristic 15. *The more times a function f is the last accessor of a memory block (p, s) output by ALLOC, the more it is likely that f is a freeing function.*

This heuristic, combined with Heuristic 14, implies that FREE should be the last function to take as a parameter the value p output by ALLOC. Algorithm 14 shows how we can compute, for a given function f, the number of times this function is the last accessor of a value output by ALLOC. This algorithm uses *Accessors*, described in Algorithm 13, which computes for a value val the list of functions f such that:

- f is ALLOC and outputs val in the corresponding return value (given by $retidx$),

- or f takes val as a parameter.

Function *Accessors(log, alloc, retidx, val)*
  Input: log a list of events sorted by event counter
  Input: $alloc$ the allocating function
  Input: $retidx$ the index of the return value of $alloc$ where the address of the allocated block is output
  Input: val the value for which we want to retrieve the accessors
  begin
    $accessors \leftarrow List()$;
    foreach *event* e *in* log **do**
      if $e.val \neq val$ **then**
        continue;
      end
      if $e.io =$ `CALL` *or* $(e.f = alloc$ *and* $e.i = retidx)$ **then**
        $accessors \leftarrow (e.f, e.i)$;
      end
    end
    return $accessors$;
  end

Algorithm 13: Compute the accessors of a value val during an execution e

7.2.6.2 General algorithm

According to these heuristics, we compute the score of every function $f \in \mathcal{F}(e)$, *i.e.* on one hand the number of times it is the last accessor of a memory block - see Algorithm 14, and on the other hand the number of its different callers. From these scores, we can output the best candidate for `FREE` in keeping with `ALLOC` - see Algorithm 15.

7.2.6.3 Discussion

Relies on `ALLOC` In the same way our detection of `ALLOC` relies on the prototype inference, `FREE` does. But more than that, the detection of `FREE` relies strongly on the detection of `ALLOC`. Indeed, our approach targets (`ALLOC`, `FREE`) as a couple, and if the first element of the couple is wrong, the couple is not relevant anymore.

```
Function NbLastAccessor(log, alloc, retidx, f, i)
    Input: log a list of events sorted by event counter
    Input: alloc the allocating function
    Input: retidx the index of the return value of alloc where the address
           of the allocated block is output
    Input: f the function to consider
    Input: i the parameter of f to consider
    begin
        nb_last_accessor ← 0;
        foreach val in ConcreteValuesR(log, alloc, retidx) do
            accessors ← Accessors(log, val);
            prev ← None;
            foreach accessor (func, param) in accessors do
                if prev = f and (func = alloc and param = retidx) then
                    nb_last_accessor ← nb_last_accessor + 1;
                end
                prev ← func;
            end
        end
        return nb_last_accessor;
    end
```

Algorithm 14: Compute the number of times a given function f is the last accessor of an allocated block through an execution e

Function *GetFree(log, alloc, retidx)*
Input: log a list of events sorted by event counter
Input: $alloc$ the allocating function
Input: $retidx$ the index of the return value of $alloc$ where the address of the allocated block is output
begin
$free \leftarrow None$;
$best\_score \leftarrow 0$;
foreach *function* f *in* $F(e)$ **do**
$score \leftarrow NbLastAccessor(log, , alloc, retidx, f)$;
if $NbCallers(log, f) > 3$ *and* $score > best\_score$ **then**
$best\_score \leftarrow score$;
$free \leftarrow f$;
end
end
return $free$;
end

Algorithm 15: Retrieve `FREE` corresponding to a given `ALLOC` from an execution e

Note that an inconsistent couple should be detected by the additionnal step to test the inferred couple - see Section 8.5.

Assume few bugs Our heuristics, and in particular Heuristic 15, assume that `FREE` is the last accessor of an allocated block. This approach supposes, as an underlying asumption, that "bugs are rare" (quoted from [CSB13] - the authors use the exact same heuristic), and especially *use-after-free* bugs. Indeed, these bugs are, by definition, bugs where a block is accessed after it has been freed. In this scenario, `FREE` is no longer the last accessor of this block. If it happens rarely enough, this does not interfere with our detection. However, if more than half freed blocks are then accessed by the same function, this could lead to an incorrect detection.

Summary of heuristics

7.3 Arity

Heuristic 1. *Inside a function, one read before write on a memory location $m \in \mathcal{M}_p(e)$ means that m is a parameter of f, but a write before read or an unused one does not allow to conclude.*

Heuristic 2. *A parameter detected for f_n and last written by f_0 is also a parameter of any f_i between f_0 and f_n in the stack of calls at the time f_n is called.*

Heuristic 3. *A return value detected for f_n and last written by f_0 is also a return value of any f_i between f_n and f_0 in the stack of calls at the time f_n returns.*

Heuristic 4. *$in(f)$ can be computed from a small number of different executions $e \in \mathcal{E}(\mathcal{B})$ such that $f \in \mathcal{F}(e)$. This number can possibly be one.*

7.4 Type

Heuristic 5. *Once a value is typed, every further occurrence of the same value has the same type.*

Heuristic 6. *A value used as a memory operand is an address.*

Heuristic 7. *For any function $f \in \mathcal{F}(e)$, an execution $\varepsilon \in e\lfloor f$, and for any $p \in in_\varepsilon(f) \cup out_\varepsilon(f)$ getting at some point the concrete value v during ε, if $type(v) = \texttt{ADDR}$, then $type_\varepsilon(p) = \texttt{ADDR}$.*

Heuristic 8. *If a parameter p of f is detected as an address in one execution $\varepsilon \in e\lfloor f$, then we deduce that its type in e is always* ADDR.

$$type_e(p) = \begin{cases} \texttt{ADDR} \text{ if } \exists \varepsilon \in e\lfloor f, type_\varepsilon(p) = \texttt{ADDR} \\ \texttt{NUM} \text{ otherwise} \end{cases}$$

7.5 Couple

Heuristic 9. *For a given execution e, given two functions f and g in $\mathcal{F}(\mathcal{E}(\mathcal{B}))$, a return value $r \in out_e(f)$ and a parameter $g \in in_e(g)$ such that $type(p) = type(r) =$* ADDR*, if r takes the concrete address value a and if p takes the same address value a further in e, we assume a data flow from r to p according to Definition 27.*

Heuristic 10. *For r a return value, and p a parameter, we can compute the rate coupling ρ of the couple (r, p) over an execution e on a sample of concrete values of p.*

7.6 Memory allocator

7.6.1 ALLOC

Heuristic 11. *The allocating function outputs as a return value the address p of the allocated block (p, s).*

Heuristic 12. *The less a function f is called by various other functions, the less it is likely that f is an allocating function.*

Heuristic 13. *The more a function f outputs new addresses (never seen before), the more it is likely that f is an allocating function.*

7.6.2 FREE

Heuristic 14. *The freeing function takes as a parameter the address p of the block (p, s) to be freed.*

Heuristic 15. *The more times a function f is the last accessor of a memory block (p, s) output by* ALLOC*, the more it is likely that f is a freeing function.*

Part III

Experiments

Chapter 8

Implementation

In this chapter, we present our implementation of the several analysis presented in Part II. For each, we give more precise details about the practical analysis, and specify several points in order to adapt efficiently to a particular target: `x86-64` binaries. We implemented every analysis described in this chapter in an open-source tool we named `scat`, that we present in the next section. `Scat` was presented at SANER 2017 [dGFM17] and is available at `https://github.com/Frky/scat`. The rest of this chapter is organized as follows: Section 8.2 and Section 8.3 present the implementation of the structure analysis, Section 8.4 presents the couple analysis and Section 8.5 presents the implementation for allocators retrieving. This implementation aims to provide a partial validation of our approach, and especially the respect of the criteria we stated in Section 4.1.3. In particular, we propose experiments to validate *accuracy* and *scalability* (see Chapter 9). However, the *universality* criterion would require to implement our approach on other architectures to be fully validated.

8.1 `Scat`

Our tool, named `scat`, implements the approaches we presented in Chapter 6 and Chapter 7. Initially, the name `scat` stood for *strong coupling, arity and type*, but because it evolved since its first version (and in particular extended to allocator retrieving and memory analysis), now it is no longer more than a wink to jazz lovers. It used to have a logo - see Figure 8.1, but since the only paper we used

Figure 8.1: Historical logo of our tool `scat`

this logo in was rejected, we gave it up (although it was probably not the reason for the reject).

In Section 8.1.1, we present the general concept of `scat` and its architecture. We then present the scope that we target with this implementation, and its limitation regarding universality - see Section 8.1.2. Finally, an overview of the technical choices we made regarding the instrumentation is given in Section 8.1.4.

8.1.1 General presentation

8.1.1.1 Usage

`Scat` is a command-line interface tool. From a prompt, one can perform online analysis, *i.e.* instrument the execution of any program and providing possible arguments. There is one command for each online analysis: `launch arity`, `launch type`, `launch couple` and `launch memalloc`. For these analysis, `scat` makes the interface with the dynamic instrumentation tool that we use: `Pin` - see Section 8.1.4.1. Every instrumented execution outputs a log file, either with the results of the inference (for arity and type) or for events to be treated in an offline analysis (for couple and memalloc). Details are given in the relative sections of this chapter for each.

`Scat` also proposes several commands to perform offline analysis from the log files produced by the online analysis. In addition to produce the results from the logged events in the case of two-steps analysis, one can use `scat` to test the results of the inference, if it can provide the sources required by the tool to construct an oracle. For instance, `scat` embeds a `C` source-code analyzer, to compare the results of structure inference with source-level information in the case of open-source

programs. This will be discussed in Chapter 9.

8.1.1.2 Architecture

The global architecture of scat is given in Figure 8.2. As mentioned before, some online analysis directly output the results (arity and type), and some others require an offline step (couple and memalloc). The online analysis is performed using Pin - see Section 8.1.4.1. To do so, we write what is called pintools, *i.e.* C++ code to specify the instrumentation to perform (including what instruction to instrument and the corresponding handler for each, the logging functions, etc.). The offline analysis is written in Python, and consists in parsing and analyzing log files produced by the online step.

8.1.1.3 Philosophy

Scat proposes an implementation of our approach with several characteristics that worth the mention. First, it presents a set of properties that are inherited from the goals we stated in Section 4.1: **scalability** and **universality**. The implementation of scat aims to target a large set of binary programs with few requirements, and with a reasonable overhead, to be able to analyze large programs. Second, and as a consequence of the approach we propose in Chapter 6 and Chapter 7, scat performs analysis **dynamically**, and each one in **one single execution**. Third, as design choices, scat is **open-source**[1] and **flexible**: anyone can add new pintools and new offline analysis, and the integration to the command line interface is documented on GitHub. .

8.1.2 Scope

We present, in this section, the scope of our implementation. In particular, we do focus on one binary architecture and one calling convention. However, our approach does extend to other architectures and our implementation could be extended to deal with them.

Remark To test our implementation, we focus on compiled programs, and that is the reason why we choose a particular calling convention. In the case of not-compiled programs, they only need a calling convention when they want to interact

[1] https://github.com/Frky/scat

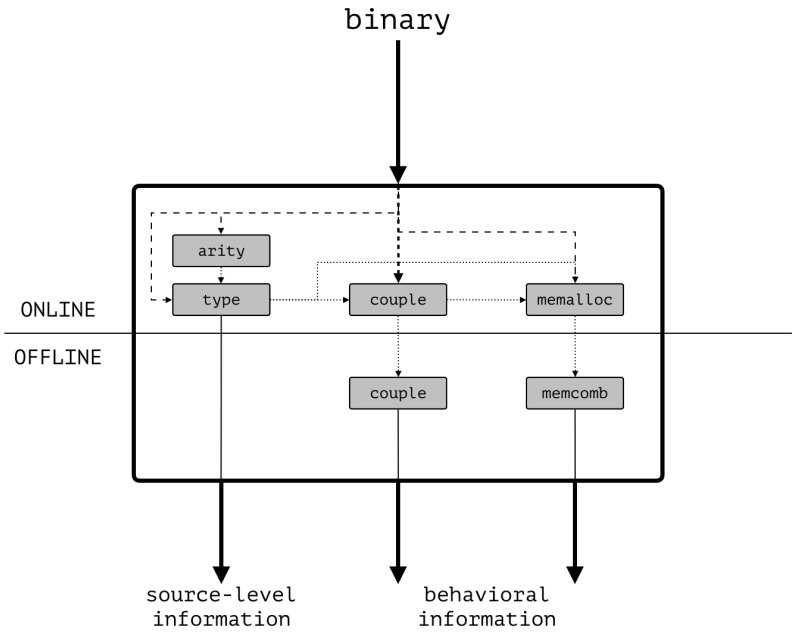


Figure 8.2: General architecture of `scat`

with libraries. Otherwise, they do not have to follow any calling convention. If we want to analyze a hand-written program that does not follow a calling convention, then the implementation of `scat` needs to be adapted.

8.1.2.1 x86-64 System V ABI

We choose to focus our implementation on x86 binaries, and in particular x86-64. This architecture is widely used in desktop computers, so we can perform analysis on a large variety of programs. Regarding the calling convention, we focus on the main calling convention on `Linux`: `x86-64 System V ABI`. The choice of targeting a `Linux` calling convention is motivated by the large set of open-source programs that we can use to test our approach and compare the results of our analysis with source-level information - see Chapter 9. In the following, we specify the main points of this calling convention, and how we use it in our implementation.

Parameters and return values This calling convention specifies memory locations that functions must use to pass (and get) parameters and return values:

- parameters - denoted as $\mathcal{M}_p$: `%rdi, %rsi, %rcd, %rdx, %r8, %r9` for integers and `%xmm0, %xmm1, ..., %xmm7` for floats,
- return values - denoted as $\mathcal{M}_r$: `%rax` for integer and `xmm0` for float.

The two lists for parameters (integers and floats) are **ordered**: if a function takes only one parameter, this parameter must hold in `%rdi`, etc. If there are more parameters than registers can hold, additional values are passed through the stack.

Note that, in this calling convention, $\mathcal{M}_p \cap \mathcal{M}_r = \emptyset$, which allows some shortcuts in the implementation: the location itself is enough to know if we are dealing with a return value or a parameter. This is a specific case, but we could imagine scenarios where it is not true. Our approach would not be limited, but our implementation would have to be adapted.

Limitations This calling convention induces, by construction, limitations in the general approach we presented. For instance, a function cannot return more than two values with this scheme. Actually, we assume in our implementation that a function returns at most one value (*i.e.*, both `%rax` and `%xmm0` cannot be used by the same function to return two values). This assumption comes from the fact we deal with compiled programs in our tests, but we could easily get rid of it to adapt our tool to functions outputting two values at the same time.

Floating values It is important to note that floating values are handled by specific registers. This means that the architecture itself is dealing with types. Consequently, we include, in the types we target, the type `FLOAT`, which is very specific: we do not use instructions but registers to conclude that a value is a `FLOAT`. This particularity is specific to the targeted architecture, and does not impact the general approach we presented in Section 6.2.

8.1.3 Identifiers from one execution to another

In our implementation, we need to have ways to identify several elements from a point of one execution to another, and from one execution to another. In particular, we need to identify functions and parameters of functions.

8.1.3.1 Functions

The problem of identifying functions we have to deal with here is very specific to our implementation. A natural way of identifying functions is by their name, but, according to the university criteria, we aim to target stripped binaries, where symbols are removed (and in particular function names). The next idea that comes is to use entry points of functions, *i.e.* the target of `CALL` instructions (or equivalent). However, because `Pin` performs *just-in-time* compilation, the entry points of functions at runtime depend on the instrumentation we perform: from one execution to another, if the instrumentation changes, the entry points of functions change. For instance, a function `f` will not have the same entry point if we perform instrumentation for arity and if we perform instrumentation for types.

Eventually, **we identify functions by a couple (`image`, `offset`)**, where `image` is the name of the image being loaded (*i.e.*, either the main program or the name of a library) and `offset` the static offset of the function in the image. This is a good identifier because 1) the name of images remain even in stripped binaries (otherwise, the dynamic loader would not know which library should be loaded), and 2) the **static** offset does not depend on the execution so it is invariant. In the following of this chapter, when we mention a function `f`, it is in fact identified by the corresponding couple that we abstract for more clarity.

8.1.3.2 Parameters

By opposition to functions, parameters have a consistent identifier: their memory location (we recall that these memory locations are expressed relatively to a base

address, so they are invariant from one execution to another). Even for stack parameters, the memory location being expressed relatively to %ebp, it is invariant.

However, for clarity, **we identify parameters by an integer** that corresponds to the order given by the calling convention we target. For instance, a parameter located in the register %rdi will be designed by the integer 1, a parameter in register %rsi will be designed by 2, etc. In addition, we use the integer 0 to design the return value of a function. We can do this because according to the calling convention we target, a function can only output one value (either through %rax or %xmm0).

8.1.4 Technical choices

In this section, we present the main technical choices we made in the implementation of scat. In particular, we present how we instrument executions, how we choose traces, and how we actually instrument calls and returns of functions.

8.1.4.1 Instrumentation: Pin

We perform dynamic instrumentation of executions using a framework developed by Intel: Pin. Through some code written in C++, called pintools, we specify what to instrument, the handlers, and actions to perform before (initialization) and after (logging) the execution of a program. We write one pintool for each analysis. The handlers, initialization and ending functions have nothing particular to be mentioned in the general case; however we give here some details about how the inspection and instrumentation is performed by and using Pin.

Before executing a binary program, using Pin's function INS_AddInstrument-Function, we call instrument_instruction for each instruction statically detected by Pin. This function takes an instruction as a parameter, and regarding the kind of instruction, add an handler on this instruction or not. This handler takes as a parameter the context of the execution when it occurs (*i.e.*, in particular, register values), and other optional arguments (for instance, we can give to the handler of a call the targeted address). For instance, Pin provides INS_IsCall which returns a boolean if the instruction is a call, either direct or indirect. If we want to instrument every direct CALL instruction, we can write something as the code presented in Listing 8.1.

```
/* If the instruction is a direct call - function provided by Pin */
```

```
if (INS_IsDirectCall(ins)) {
  /* Get the targetted address - function provided by Pin */
    ADDRINT addr = INS_DirectBranchOrCallTargetAddress(ins);
    /* Get the function being called from the targetted address */
    FID fid = fn_lookup_by_address(addr);
    /* Insert handler - function provided by Pin */
    INS_InsertCall(ins,
                IPOINT_BEFORE,
                /* Handler to be executed */
                (AFUNPTR) fn_call,
                /* Context of the call, ie reg values etc. */
                IARG_CONST_CONTEXT,
                /* Function being called */
                IARG_UINT32, fid,
                IARG_END);
}
```

Listing 8.1: Example of an instrumentation using `Pin`

The other interesting inspection functions we use in our pintools are:

- `INS_RegRContain` - return `true` if the instruction performs a read on a given register,
- `INS_RegWContain` - return `true` if the instruction performs a write on a given register,
- `INS_IsStackRead` - return `true` if the instruction performs a read of the stack,
- `INS_IsRet` - return `true` if the instruction is a return,
- `INS_MemoryOperandCount` - return the number of memory operands of the instruction,
- `INS_MemoryOperandIsWritten` - return `true` if the memory operand given in parameter is written.

8.1.4.2 Number of calls to conclude

For the structure analysis we implemented, *i.e.* for arity and type detection, we use a parameter `MIN_CALLS`[2], that can be specified before the instrumented execution. This parameter states the number of times a given function must be called -at

[2]Actually, in the case of type detection, it is named `MIN_VALS`, but it has the same role - see Section 8.3.5.1

least- during the instrumentation if we want a result. In other words, for a function called less than MIN_CALLS during an execution e, we will not conclude on its arity (or type). On the other hand, we can conclude for every function that is called more than MIN_CALLS. A high value of MIN_CALLS means that we will conclude on a small number of functions (especially for short executions); however the results should be more accurate. The influence of this parameter on both the number of functions inferred and the accuracy is discussed in Sections 9.2.2 and 9.3.2.

8.1.4.3 Call and ret instrumentation

In Section 5.2.1.1, we proposed a semantic definition of CALL and RET, independently from an instruction set. Here, we specify this definition to x86-64 instruction set.

CALL To instrument function calls, we first use the Pin inspection function named INS_IsCall which includes every well-identified CALL instruction (*i.e.*, direct call from register, indirect call from register, direct call from address, etc.: all these instructions have a different opcode in x86-64). These instructions correspond to the CALL as we defined it in Definition 7: it saves the return address on the stack and modifies the value of pc to execute the targeted address.

In addition, and to include jump-based calls, we also instrument every indirect branching that is not a call (if it is a call, then it is instrumented from the first point). We use INS_IsIndirectCallOrBranch to perform this. For this instrumentation, we do not check if a return value has been saved before the jump. This is an empirical deduction: among the calls that we miss with the INS_IsCall instrumentation, a large proportion are calls performed through indirect branching, and almost every indirect branching corresponds to a call in a compiled code, in comparison with the source-level code.

RET To instrument returns, we use two mechanisms. First, we instrument every RET function of the instruction set using INS_IsRet. This is the main and classical way to handle returns. However, and especially regarding optimized compiled code, one RET instruction can correspond to multiple functions in the same time (*tail call* mechanism). To handle this, we use a stack of calls that we present in the next section, which is updated at each call and return. When a function returns, every function in the stack of calls that was called after this function and that have not returned yet is also considering as returning.

8.1.4.4 Stack of calls

For every instrumentation of our implementation, we keep track of the stack of calls that we update at each call and return we instrument. For scalability, and in order to instrument large binaries that use deep recursivity (*e.g.* `git`), we implement a particular call stack we named *Hollow Stack*. Moreover, we add a new feature that we need for descendant propagation of parameters (see Section 6.1.4.3) that we named *shadow of the stack*. The call stack we use in the rest of this chapter implements both features.

Hollow Stack The aim of this design named *Hollow Stack* is to keep a good sampling of the calls that occurred and have not returned yet, even when it is not possible (or very inefficient) to keep every single call in the stack. In general, even in large binaries, the call stack is not really deep, except for programs using recursion: this leads to performance and memory issues. The *Hollow Stack* behaves exactly like a straightforward fixed-size stack until a given limit size is reached. In this case, it will start to discard elements in the middle. The intuition is that with a reasonably-sized stack[3], overflow only occurs in cases of heavy recursion. When this happens, the stack is filled with a lot of redundant calls in its middle while the bottom and the top will contain relevant calls (head and tail of the recursive calls chain). The principle of this stack is shown in Figure 8.3.

Shadow of the call stack We propose to add another functionality to our call stack, we name *shadow of the stack*. When an element is popped from the stack, the head of the stack is updated accordingly, but the elements are not actually deleted. The actual deletion is performed when a push occurs. Therefore, it is possible to access every element that was popped after the last push. What we name *shadow of the stack* is the set of elements that were popped but still accessible in fact. Figure 8.4 illustrates this idea. A concrete example of this is what actually occurs in the stack of a binary during the execution: an instruction `POP` moves the stack pointer but does not actually erase the data.

[3]For example, the number of functions inside the executable can be used as a large upper bound

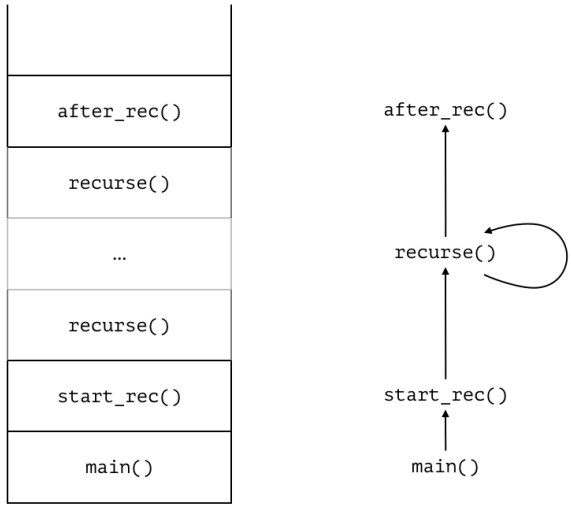


Figure 8.3: Recursive calls chain and the corresponding hollow stack

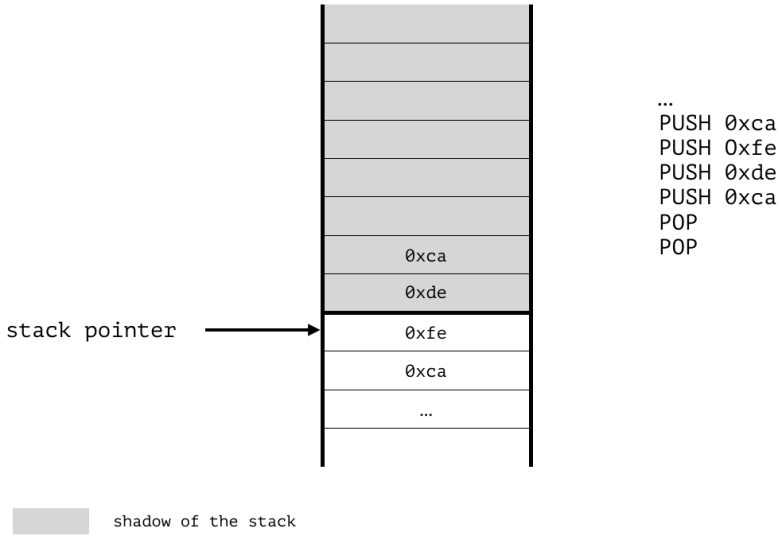


Figure 8.4: Illustration of the shadow of the stack on an example

8.2 Arity

In this section, we present the specificities of our implementation to retrieve arity of functions. This implementation takes some liberties regarding the approach we presented in Section 6.1. We present an evaluation of it in Section 9.2.

8.2.1 Combined analysis

In Section 6.1, we proposed an approach in two steps: an online instrumentation step and an offline inspection step. In our implementation, we actually do both steps at once: deductions on arity are made during the instrumentation, and this only step directly outputs the arity results. The choices we made that illustrate this distinction with the approach we described previously are detailed in the rest of this section.

8.2.2 Practical heuristics

In addition to heuristics we presented in Section 6.1.4, which were relative to the approach, we use some heuristics relative to the implementation. These heuristics do not improve the theoretical approach, but may correct some misdetections due to an imperfect instrumentation. In particular, we assume that we might miss some reads, writes, calls and returns during our implementation.

8.2.2.1 Parameters and returns

As mentioned in Section 8.1.2, some specific memory locations hold parameters. To pass a parameter, the caller must write its value at one of these memory locations, and the callee fetches it after the `CALL`. As a reminder, if a function f writes at some memory location $m \in \mathcal{M}_p$, g is called later (but before f returns) and reads at memory location m, and if m was not overwritten in between, then we deduce that m holds a parameter of g. However, we claim that this only holds if no function (neither f nor any other one) returns in between. Indeed, a function has no obligation to preserve values in registers, and therefore we cannot assume that m would not have been overwritten (maybe it has not been during one particular observed execution, but we cannot assume this is the general case). This leads to the following heuristic:

Practical heuristic 1. *For a function f accessing a memory location $m \in \mathcal{M}_p$, if a* `RET` *occurred since the last write of m, then m cannot hold a parameter of g.*

An example of such a situation (artificially built) is given in Listing 8.2. In principle, such a scenario should not occur with compiled code: `f` should not be accessing `%edi` in this way after `g` returned. However, what can happen is that we miss some instructions, for instance a write of `f` in `%edi` after `g` returned. This heuristic is thus mainly used to correct some errors due to imperfect instrumentation.

```
f:
    PUSH    %ebp
    MOV     %esp, %ebp
    ...
    // f calls g
    CALL g
    ...
    // then f reads %edi
    MOV     %edi, %eax
    ...
g:
    PUSH    %ebp
    MOV     %esp, %ebp
    ...
    // in between, g overrides %edi
    XOR     %edi, %edi
    ...
    RET
```

Listing 8.2: Example of a register being read after a return that may lead to a misdetection

8.2.2.2 Return values and calls

In the same way, we cannot assume that a memory location $m \in \mathcal{M}_r$ is preserved by other function calls. Therefore, a `CALL` between the write of m and its read prevent the detection of a return value:

Practical heuristic 2. *For a function f accessing a memory location $m \in \mathcal{M}_p$, if a* `CALL` *occurred since the last write of m, then m cannot hold a return value of g.*

8.2.3 Instrumentation

To perform arity analysis, as mentioned in Section 6.1.3.3, we need to instrument two types of event: function calls and returns, and location accesses. Function calls and returns are instrumented as presented in Section 8.1.4.3. We only add a few specific operations in the corresponding handlers - see next section. Regarding location accesses, we proceed as follows: we provide two lists, one for parameter locations and one for return value locations; and each instruction that either reads or writes one of these locations is instrumented - handlers for each are detailed in next section. These lists correspond to the memory locations specified by the calling convention to pass parameters and return values - see Section 8.1.2. The specific case of stack parameters is discussed in Section 8.2.4.3.

8.2.3.1 Data

Our instrumentation uses the several global variables:

- `ec`: an event counter incremented by each handler,
- `sc`: stack of calls,
- `ssp`: stack of `%esp` values - this is required to detect stack parameters,
- `last_call`: value of the event counter when the latest call occurred,
- `last_ret`: value of the event counter when the latest return occurred,
- `last_write`: array where, for each memory location `m` monitored for parameters or return value, `last_write[m]` is the event counter of the latest event that performed a write at memory location `m`,
- `nb_calls`: array to store the number of calls of each function,
- `nb_detection`: 2D array, where `nb_detection[f][m]` is the number of executions of the function f where the memory location $m \in \mathcal{M}_p$ was detected as a parameter (or a return value if $m \in \mathcal{M}_r$).

8.2.3.2 Instrumented schemes

For arity detection, we instrument any instruction that corresponds to one of the following schemes:

- a function `CALL`,
- a function `RET`,
- an access to a memory location of $\mathcal{M}_p$ or $\mathcal{M}_r$.

8.2.3.3 Handlers

For each of these instrumented schemes, we perform some actions in the corresponding handler, that aims to prepare the final detection of arity: in addition to logging events (as presented in Section 6.1.3.3), some information relative to inspection are directly treated at runtime.

Functions

Call Each time a function `f` is called, we perform the following actions:

- push `f` to the call stack `sc`: `sc.push(f);`
- push `%esp` to the sp stack `ssp` - note that it is the old value of `%esp`, *i.e.* the value before the call that is pushed: `ssp.push(%esp);`
- increment the number of calls to `f`: `nb_calls[f]` $\leftarrow$ `nb_calls[f] + 1;`
- save the event counter of the last call: `last_call` $\leftarrow$ `ec;`
- increment the event counter: `ec` $\leftarrow$ `ec + 1.`

Return When a function `f` returns, we perform the following actions:

- pop `f` from the call stack `sc`: `sc.pop();`
- pop `%esp` from the sp stack `ssp`: `ssp.pop();`
- save the event counter of the last return: `last_ret` $\leftarrow$ `ec;`
- increment the event counter: `ec` $\leftarrow$ `ec + 1.`

Parameters

On read For a read access to a memory location `m` of $\mathcal{M}_p$, we do:

- if `last_ret > last_write[m]`, do nothing - this means that a function has returned since the last write of `m`;
- for each function `f` in the call stack, if the call to `f` occurred after `last_write[m]`, increment the corresponding number of detection: `nb_detection[f][m]` $\leftarrow$ `nb_detection[f][m] + 1`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

On write For a write access to a memory location `m` of $\mathcal{M}_p$, we do:

- update the last write of `m`: `last_write[m]` $\leftarrow$ `ec`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

Return value

On read When a memory location `m` for return value, *i.e.* in $\mathcal{M}_r$, is read, we do the following:

- if `last_call > last_write[m]`, do nothing - this means that a function has been called since the last write of `m`;
- for each function `f` in the shadow of the call stack, if the return of `f` occurred after `last_write[m]`, increment the corresponding number of detection: `nb_detection[f][m]` $\leftarrow$ `nb_detection[f][m] + 1`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

On write When a memory location `m` for return value, *i.e.* in $\mathcal{M}_r$, is written, we do the following:

- update the last write of `m`: `last_write[m]` $\leftarrow$ `ec`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

8.2.4 Detection

From the data we collect during the execution, we present here our method to retrieve arity of functions.

8.2.4.1 Thresholds

In addition to the global parameters presented in Section 8.1.4, and in particular `MIN_CALLS`, the arity detection relies on two other adjustable parameters, one to conclude on parameters and one to conclude on return values. The influence of both is evaluated in Section 9.2.

Because executions of a given function are not necessarily sequential, several paths may lead to use or not some parameters the function takes. This would lead to false negatives (*i.e.*, parameters that we miss). On the other hand, if our instrumentation misses some writes of memory locations, we may end up with false positives (*i.e.* detecting parameters that do not exist). This is the reason why we use two thresholds: `PARAM_THRESHOLD` and `RET_THRESHOLD`. These thresholds are used to conclude on the arity based on the rate of detection of each parameter (resp. return value) relatively to the number of executions of the function.

8.2.4.2 Deduction of arity

At the end of the execution, the two arrays `nb_calls` and `nb_detection` contain respectively the number of calls of each function and the number of times a given parameter/return value has been detected for a given function. Our algorithm to compute arity, from these data collected at runtime, is given in Algorithm 16. Actually, in addition to the number of parameters of each function, we keep the corresponding memory locations, as this information is needed for further analysis (and in particular, type analysis).

8.2.4.3 Stack parameters

Identification and location As for other memory operands, we instrument each read and write for which the memory location is computed relatively to `%esp` or `%ebp`. Reads and writes are handled in the exact same way as for registers, except that the memory location is identified by an offset relatively to `%ebp`, and it is the

reason why we need to keep a stack of values of %ebp in addition to the call stack[4].

Size Stack parameters are harder to deal with than register parameters. Indeed, if we can elude the size of register parameters, it is very important when it comes to the stack: from eight bytes on the stack, we can hardly conclude on the number of parameters (two `int` ? eight `char` ? etc.). The strategy we adopt is to reduce a stack access of a memory location to its first byte. In other words, if at some point `% ebp - 4` is read, the number of bytes actually read does not matter: we only log a read access to `% ebp - 4`. The idea behind this is that if a stack parameter is 4-bytes long, it will never be accessed from the second, third or fourth byte.

8.3 Type

In this section, we present our implementation to retrieve types of parameters and return values.

8.3.1 Context

For the recall, we defined in Section 6.2 the subset of types we were interested in:

$$\mathcal{T} = \{\texttt{NUM}, \texttt{ADDR}\}$$

We defined ADDR as the subset of the following abstract instructions:

$$\texttt{ADDR} = \{\texttt{LOAD}_1, \texttt{STORE}_\texttt{r}\}$$

In this implementation, we extend ADDR to any memory operand of any instruction of the x86-64 instruction set - see Section 8.3.5.3.

In addition, we mentioned in the previous section that floating values are already typed at assembly level because they use specific registers. Thus, in the scope of this implementation, we actually distinguish three types: FLOAT, ADDR and INT, where FLOAT are detected at the previous step using register-based deductions, and ADDR are detected as detailed in the following of this section, using instruction-based deductions - as presented in Section 6.2.

[4] As explained in Section 8.3.5.2, we actually store the value of %esp, but just after a call, the value of %esp is copied into %ebp, so it does correspond to the value of the %ebp of the function being called

8.3.2 Base point

The type detection is based on the arity results. In particular, we need, from the previous inference, the following data for each function `f`:

- the location of each non-float parameter of `f`,
- the location of each float parameter of `f`,
- whether `f` outputs a value or not,
- and if so, the memory location corresponding to its value.

In other words, we need more than the strict arity, but these information are retrieved from the arity detection, so we do not need more than what can be output by the implementation presented in Section 8.2.

In the following, we abstract the memory location when we denote by `p` a parameter of a function `f`, but in practice we use this memory location to get the concrete value of `p` when `f` is called. We denote by `P[f]` the list of parameters of `f` that are not floating parameters, and `R[f]` the information about its return value: `None` if no return value for `f` was detected during arity inference, `FLOAT` if it was detected as a floating return value and `NUM` otherwise. We recall that `FLOAT` parameters can be deduced from the memory location itself.

8.3.3 Combined analysis

As for the arity inspection, we take some liberties regarding the two-steps approach we presented in Section 6.2. In particular, we deduce some elements from the instrumentation at runtime, and instead of outputting a log file, we directly output the inference results. The main advantage of this is to not store large log files, and to speed up the approach with runtime deductions. However, we could not do this in the general case: we use practical heuristics that are based on the scope of the concrete architecture we target.

8.3.4 Practical heuristic

In the previous section, we presented practical heuristics to minimize the errors due to an imperfect instrumentation - see Section 8.2.2. Here, we present practical heuristics for another purpose. Instrumenting every memory access, and keep

track of every single value detected as an address, would be very heavy and hardly scalable. Our practical heuristic aims to speed up the instrumentation and keep scalability.

Practical heuristic 3. *The address space is continuous, and therefore a value* `v` *that is between two addresses* a_1 *and* a_2 *is an address.*

This means that, instead of keeping a list of concrete values that have been detected as addresses, we only need to keep track of the lower and the upper bound.

8.3.4.1 Regions

To do so, we implement what we call *regions*. A region is characterized by two attributes and two methods:

- `low`: the lower bound of the region,
- `high`: the higher bound of the region,
- `contains(addr)`: return `true` if and only if `addr` is between `low` and *high*,
- `extend_to(addr)`: update bounds to include `addr` - if `addr < low` then `low` takes the value of `addr`, and if `addr > high` then `high` takes the value of `addr`.

In other words, a region is an interval of addresses that we can update when we detect a new address. For type detection, we use two regions: the **address space**, that we denote by `addr_space`, and which aims to include the heap and the stack at runtime; and the **data section**, denoted by `data_space`, which aims to represent the memory area where static data needed by the program at runtime are stored. The address space region is initialized empty, with `low = 0xFFFFFFFFFFFFFFFF` and `high = 0`. It aims to be updated, during the execution, every time a new address is detected. On the other hand, the data section region is initialized according to the data segment described in the header of the binary, and is invariant during the execution.

8.3.5 Instrumentation

8.3.5.1 Parameters

The detection of types is parameterized by two values: `MIN_VALS` and `MAX_VALS`. `MIN_VALS` has the same role as `MIN_CALLS` in the arity detection: it represents the minimum number of values we need to conclude on the type of a given parameter. However, to avoid collecting a too large amount of data, we also add a maximum limit, `MAX_VALS`, from which we stop collecting additional values. In other words, during the instrumentation, we collect at most `MAX_VALS` values for each parameter `p` of each function `f`, and during the detection we only conclude on the type of a parameter if we collected at least `MIN_VALS` values.

8.3.5.2 Data

Our implementation relies on the following global variables:

- `ec`: an event counter incremented by each handler,
- `sc`: stack of calls,
- `addr_space`: a region representing the address space, as described in Section 8.3.4.1,
- `vals`: a 2D array, where `vals[f][p]` is a list of concrete values for the parameter `p` of `P[f]` ; note that `vals[f][0]` is a list of concrete return values of `f`.

Remark We do need the call of stack to have information about which function is returning. Indeed, when a `CALL` is performed, the operand tells us which function is being called. However, when a `RET` occurs, there is no way to know at which function it corresponds, except by keeping a consistent call of stack.

8.3.5.3 Instrumented schemes

For type detection, we instrument three types of schemes: two relative to functions and one relative to memory accesses:

- any function call,

- any function return,
- any read or write to a memory location.

8.3.5.4 Handlers

For each instrumented scheme, an handler is called to perform the relevant actions we need for type detection.

Functions

Call When a function `f` is called, we perform the following actions:

- push `f` to the call stack `sc`: `sc.push(f)`;
- for each parameter `p` of `P[f]`, if `len(vals[f][p]) < MAX_VALS`, then we add the concrete value `v` of `p` to the list of collected values: `vals[f][p].add(v)`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

Return When a function `f` returns, we perform the following actions:

- pop `f` from the call stack `sc`: `sc.pop()`;
- if `R[f]` is `NUM`, and if `len(vals[f][0]) < MAX_VALS`, then we add the concrete value `v` of the return value to the list of collected values: `vals[f][0].add(v)`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

Memory accesses When an address `a` is used as a memory operand, we do:

- if `a` is not in the address space inferred so far, extends the address space to include `a`: `addr_space.extend_to(a)`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

In other words, the only thing we do when an address is observed is to update the bounds of the address space to include this new address (if needed).

8.3.6 Detection

8.3.6.1 Thresholds

We base our detection of types on two thresholds: `MIN_VALS` and `ADDR_THRESHOLD`. They are parameters of the implementation, and their impact is evaluated in Section 9.3.2.2.

`MIN_VALS` We presented it in Section 8.3.5.1: it states the minimal number of values we need to deduce the type of a parameter. If, for a given parameter, we did not collect at least `MIN_VALS` concrete values during the execution, then we do not infer its type. The higher this value is, the more accurate is the result, but the less parameters we are able to infer.

`ADDR_THRESHOLD` This rate is the minimum proportion of addresses a parameter should take as concrete values to be typed as an address. Indeed, we could have a few unfortunate collisions between addresses and integers (and therefore, an `INT` parameter could take a value that is inferred as an `ADDR`). The `ADDR_THRESHOLD` value states the maximum number of unfortunate collisions that are still acceptable to deduce a given parameter is an address.

8.3.6.2 Deduction of types

At the end of the execution, we have the bounds of the inferred address space `addr_space`, concrete values of parameters and return values of each function in `vals`. For each concrete value of a given parameter `p`, we compute the number of times it is detected as an address, relatively to the total number of values. If this rate is above `ADDR_THRESHOLD`, we deduce that `p` is an `ADDR` parameter. This is detailed in Algorithm 17. This algorithm outputs, for a given function `f`, a type for each of its parameters in an array, plus a value for the type of its return value.

8.3.6.3 Output

At this point, `scat` outputs `C`-like undertyped prototypes of functions - see Listing 8.3 for an example[5]. Note that this is the result of only two instrumented executions of a binary, possibly stripped.

[5] In this example, and in the following ones in Chapter 9, we work on unstripped binaries for more clarity, but our tool can deal as well on stripped binaries

```
scat > display grep type
addr fts_alloc(addr, addr, int);
void pr_sgr_start_if(int);
int kwsexec(addr, addr, int, addr);
void setbit(int, addr);
void print_sep(int);
addr strcmp(addr, addr);
int tr(int, int);
int EGexecute(addr, int, addr, int);
int undossify_input(addr, int);
addr _dl_fixup(int, int);
...

Information about inference
| Last inference:           2016-07-21 14:53:05
| Total functions infered:  38
```

Listing 8.3: Example of the display of C-like prototypes using `scat`

8.4 Coupling

8.4.1 Context

The goal of this implementation is to retrieve address-based couples, as defined in Section 7.1.2.2. This detection is relative to an execution of the program, and parameterized by a coupling rate ρ. We only target coupling involving addresses, so the instrumentation only targets functions that use addresses (either in parameter or as a return value).

8.4.2 Base point

To retrieve couples, we need the results of type inference. Actually, we require to know, among the parameters of a given function `f`, which is an address, and if `f` outputs an address or not. We denote by $\mathtt{P}_a\mathtt{[f]}$ the set of address parameters of `f`. Note that in fact $\mathtt{P}_a\mathtt{[f]}$ contains memory locations corresponding to these parameters, for the couple analysis to be able to get concrete values at runtime. In addition, we denote by $\mathtt{R}_a\mathtt{[f]}$ a boolean to indicate if `f` returns an address (`true`) or not (`false`). These data can be reconstructed directly from the output of the type analysis for each function that has been inferred.

8.4.3 Two-steps analysis

By opposition to first two implementations we presented in this chapter, our implementation of the coupling analysis is a two-steps analysis, as described in Section 7.1. During the first step, we instrument the execution and log the relevant information we need to retrieve couples. This step is presented in Section 8.4.4. The second step consists in an offline parsing of the logs and a computation of coupling rates to retrieve ρ-couples. The implementation of this step is given in Section 8.4.5.

8.4.4 Instrumentation

8.4.4.1 Parameters

As for type detection, we base our instrumentation on two parameters: `MIN_VALS` and `MAX_VALS`. These two parameters have the same role as presented in Section 8.3.5.1. `MIN_VALS` is used as a threshold to conclude or not on coupling (if there is not enough values, we cannot conclude), whereas `MAX_VALS` is used at runtime to stop the accumulation of values when we consider that we have enough of them.

8.4.4.2 Events

For each of the two events presented in Section 3, and that we need to perform coupling detection, we explicit here the concrete elements that are logged.

Call For each call of `f`, and for each parameter `p` of P_a`[f]`, we log:

- `ec` - the event counter when the call occurs;
- `fid` - the identifier of the function `f` that is called;
- `pid` - the identifier of the parameter `p` of `f`;
- `v` - the concrete value of the parameter.

Return Similarly, for each return of `f`, we log:

- `ec` - the event counter when the call occurs;
- `fid` - the identifier of the function `f` that is called;

- `rid` - a special identifier to indicate this is a return value of `f`;
- `v` - the concrete value returned by `f`.

8.4.4.3 Instructed schemes

For coupling, we instrument two instruction schemes, and although in the previous analysis we instrumented every single call and return, here we filter the functions to be instrumented. We instrument the following schemes:

- every call to `f` if `f` takes at least one `ADDR` parameter,
- every return from `f` if `f` outputs an `ADDR` return value.

8.4.4.4 Handlers

Call When a function `f` is called, we perform the following:

- push `f` to the call stack `sc`: `sc.push(f)`;
- increment the number of calls to `f`: `nb_calls[f]` $\leftarrow$ `nb_calls[f] + 1`;
- for each parameter `p` of P_a`[f]`, if `len(vals[f][p]) < MAX_VALS`, log the corresponding event;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

This handler aims to collect concrete values for each address parameter of `f`, up to `MAX_VALS` values.

Return When a function `f` returns, we perform the following:

- pop `f` from the call stack `sc`: `sc.pop()`;
- if R_a`[f]` is true, then log a return event with the concrete value `v` of the return value of `f`;
- increment the event counter: `ec` $\leftarrow$ `ec + 1`.

This handler aims to collect all concrete address values output by `f` during the execution.

8.4.5 Detection

8.4.5.1 Parameters and thresholds

The detection of couples is based on the rate ρ that is a parameter of the analysis: we only output couples that have a coupling rate equal or above ρ. In addition, and as mentioned earlier, we use the threshold `MIN_VALS` to determine whether the coupling rate of a given couple (`f`, `g`) can be computed: if the number of parameter values of `g` is not enough (*i.e.* if it is less than `MIN_VALS`), we skip it.

8.4.5.2 Deduction

From the log we obtained during the instrumented execution, we need to reconstruct a collection of values, stamped with the event counter (to keep track of the chronology) for each parameter of each function. This is performed in Algorithm 18. Thus, we can compute the coupling rate for each possible couple, and only output the relevant ones (*i.e.* the ones with a coupling rate above ρ). Algorithm 19 presents the implementation.

8.5 Allocators

This section presents our implementation of allocator retrieving. As for coupling, the actual implementation is really close to the approach we presented in Section 7.2.4.

8.5.1 Context

We aim to retrieve the main allocator of a program during one execution. By "main one", we mean the one that is the most frequently used. For the recall, we target allocators as defined in Section 5.4, *i.e.* we need to retrieve two functions: the allocating function, that we generally denote by `ALLOC`, and the freeing function that we denote by `FREE`. A given program could embed several allocators, *i.e.* several couples `(ALLOC, FREE)`, either within several layers or even completely independent ones. In the case of a multiple-layers allocator, it is easy, from the detected layer (*i.e.*, the most frequently used one), to retrieve the others layers by tracking addresses it produces. In the case of several independent allocators, our implementation can be extended.

8.5.2 Base point

For this analysis, we require the results of prototype analysis (*i.e.*, arity and type retrieving), plus the results of coupling analysis. From prototype analysis, we actually need the set $\mathtt{P}_a\mathtt{[f]}$ of parameters that are addresses, for each function `f`, plus the boolean $\mathtt{R}_a\mathtt{[f]}$ which indicates if `f` returns an address or not. This requirement is the same as for coupling analysis. From the coupling analysis, we need a list $\mathtt{C}_\rho$ of tuples `(f, g, i)` such that the output of `f` is ρ-coupled with the $\mathtt{i}^{th}$ parameter of `g`. Here again, ρ is a parameter of the implementation. We denote by $\mathcal{F}_\rho$ the set of functions that are involved in a couple of $\mathtt{C}_\rho$, either as a left or as a right operand of the couple.

8.5.3 Practical heuristics

To perform the analysis on concrete binaries, we need to specify general heuristics proposed in Section 7.2.5.1. In particular, we make a strong assumption relative to the prototype of `ALLOC`.

8.5.3.1 Memory bloc and address

Because we work with `ADDR` concrete values, we reduce a memory block to the address of its first cell in our implementation. This means, in particular, that two different addresses are seen as two blocks. The problem it raises, when one wants to retrieve an allocator, is that it is hard to distinguish new addresses corresponding to new blocks and new addresses corresponding to an existing block (beginning of the block plus an offset). In particular, Heuristic 13 does not work anymore, because a function that returns addresses within a block will be seen as a producer of addresses.

8.5.3.2 Prototype of `ALLOC`

To fix this, we add a practical heuristic on the prototype of `ALLOC`. If functions can easily produce new addresses with an offset from an existing address, only `ALLOC` can produce new addresses from scratch, *i.e.*, without taking an address as a parameter.

Practical heuristic 4. `ALLOC` *does not take an address as a parameter.*

8.5.4 Instrumentation

8.5.4.1 Parameters

The only parameter of this instrumentation is ρ, which is in fact inherited from the parameter of coupling analysis: as we base our allocator detection on ρ-coupled functions, we must have perform the coupling step with the same ρ before. However, as we need to collect every address value involved, there is no parameter such as `MIN_CALLS` or `MAX_VALS`.

8.5.4.2 Handlers and logging

The instrumentation of the execution for allocators detection is very similar to the one for coupling analysis: the same handlers are used, and the same data is logged. The only two differences are the following.

First, we do not instrument the same functions. For coupling, we instrument *every function* that deals with addresses. Instead, here we only instrument `CALL` and `RET` for functions that are involved in a ρ-couple, according to the results of Section 8.4. **Second**, by opposition with coupling detection, for allocators we need to collect *every concrete value* for parameters we are interested in (*i.e.*, `ADDR` parameters involved in a ρ-couple).

In conclusion, the target of our inspection is every function `f` $\in \mathcal{F}_\rho$*, and for each of these functions we collect every* `ADDR` *data, either in parameter or as a return value.*

8.5.5 Detection

From the logged events, we perform an offline analysis to retrieve the allocator in two steps: first, we find the best candidate for `ALLOC`, and from it we retrieve the best candidate for `FREE`. We propose, in Section 9.5, a metric to evaluate the relevancy of the output couple (`ALLOC`, `FREE`). If the consistency is below a given threshold, we can try with the second best candidates, etc.

8.5.5.1 Implementation of the address table

To retrieve `ALLOC` and `FREE`, we need to track addresses when they are used.

`AddrTable` For `ALLOC` we need to be able to know if a given address was seen before, and to mark a new address as *seen*. To to this efficiently, we implement an address table, `AddrTable`, which is basically a hashtable with three main methods:

- `add(a)` - add the address `a` in the table,
- `contains(a)` - return `true` if and only if `a` was added to the table before,
- `items()` - return the list of addresses that correspond to an entry in the table.

`AddrTableDic` For `FREE`, we do need to know if an address was seen before, but in addition we need to store a list of accessors for each address. We implement a variant of `AddrTable`, named `AddrTableDic`, where each address is associated with a list of accessors, initially empty. In addition of the two methods inherited from `AddrTable`, a third method allows to add a function to the list of accessors of an address:

- `add_accessor(a, f)` - add `f` to the list of accessors of `a` (if `a` is not in the address table, it is added).

8.5.5.2 Deduction

We present, in Algorithms 21 and 23, versions of the algorithms presented in Section 7.2 adapted to the notations of the implementation and dealing with the parameters, but the logic of the algorithm is unchanged. This implementation relies, for `ALLOC`, on the dictionary `callers`, computed by the function `ComputeCallers` presented in Algorithm 20 ; and for `FREE` on the list of accessors of each address `accessors` computed in Algorithm 22. We also use data structures we implemented to improve the efficiency of the analysis, and in particular the address table to keep track of the new addresses.

```
Function Arity(nb_calls, nb_detection, f)
    Input: nb_calls the number of calls of each function
    Input: nb_detection the number of detection of each memory location
           for each function
    Input: f the function for which we compute the arity
    begin
        if nb_calls[f] < MIN_CALLS then
            /* The number of calls to f is too small to be able
               to conclude                                      */
            return n.c., n.c.;
        end
        nb_in ← 0;
        foreach memory location m ∈ M_p do
            rate ← nb_detection[f][m] / nb_calls[f];
            if rate > PARAM_THRESHOLD then
                nb_in ← nb_in + 1;
            end
        end
        nb_out ← 0;
        foreach memory location m ∈ M_r do
            rate ← nb_detection[f][m] / nb_calls[f];
            if rate > PARAM_THRESHOLD then
                nb_out ← nb_out + 1;
            end
        end
        return nb_in, nb_out;
    end
```

Algorithm 16: Get the arity of function `f` from one execution

```
Function Type(addr_space, data_space, vals, f)
    Input: addr_space the region corresponding to the inferred address space
    Input: data_space the region corresponding to the data section
    Input: vals 2D array with concrete values of parameters and return values
    Input: f the function for which we compute type of its parameters
    begin
        type ← Array() ;
        foreach parameter p of f do
            if p not in P[f] then
                type[p] ← FLOAT ;
                continue;
            end
            nb_addr ← 0;
            foreach value v of vals[p][v] do
                if addr_space.contains(v) or data_space.contains(v) then
                    nb_addr ← nb_addr + 1 ;
                end
            end
            rate ← nb_addr / len(vals[f][p]);
            if rate > ADDR_THRESHOLD then
                type[p] ← ADDR ;
            else
                type[p] ← INT ;
            end
        end
        /* Dealing with the return value                                  */
        if R[f] is FLOAT then
            ret_type ← FLOAT ;
        else
            nb_addr ← 0;
            foreach value v of vals[f][0] do
                if addr_space.contains(v) or data_space.contains(v) then
                    nb_addr ← nb_addr + 1 ;
                end
            end
            rate ← nb_addr / len(vals[f][0]);
            if rate > ADDR_THRESHOLD then
                ret_type ← ADDR ;
            else
                ret_type ← INT ;
            end
        end
        return type, ret_type;
    end
```

Algorithm 17: Get the type of parameters for function `f` from one execution

```
Function CollectValues(events)
    Input: events a list of events
    begin
        /* vals is a 2D array of lists, where vals[f][p] is a
           list of concrete values for parameter p of function
           f                                                    */
        vals ← 2DArray() ;
        foreach event e in events do
            vals[e.fid][e.pid].add(e);
        end
        return vals ;
    end
```

Algorithm 18: Get concrete values for each parameter (and return value) of functions from the logged events

```
Function Couples(vals)
    Input: vals a 2D array where vals[f][p] is a list of concrete values
           for parameter p of function f
    begin
        couples ← Array();
        foreach function f in vals.indexes() do
            if not R_a[f] then
                continue;
            end
            foreach function g in vals.indexes() do
                foreach parameter p in P_a[g] do
                    if len(vals[g][p]) < MIN_VALS then
                        continue;
                    end
                    nb ← 0 ;
                    foreach value v in vals[g][p] do
                        if v in vals[g][0] then
                            nb ← nb + 1 ;
                        end
                    end
                    if nb / len(vals[g][p]) > ρ then
                        couples.add(f, g, p);
                    end
                end
            end
        end
        return couples ;
    end
```

Algorithm 19: Get concrete values for each parameter (and return value) of functions from the logged events

```
Function ComputeCallers(events)
    Input: events a list of events
    begin
        call_stack ← CallStack();
        callers ← Dict();
        foreach event e in events do
            if e.is_call() then
                caller ← call_stack.top();
                call_stack.push(e.fid);
            else
                call_stack.pop();
                caller ← call_stack.top();
            end
            if caller not in callers[e.fid] then
                callers[e.fid].add(caller);
            end
        end
        return callers ;
    end
```

Algorithm 20: Get the callers of every function from the logged events

```
Function GetAlloc(events, callers)
    Input: events a list of events
    Input: callers a dictionary, where for each function f, callers[f] is
           a list of all the callers of f during the execution
    begin
        nb_new_addr ← Dict();
        addr_seen ← AddrTable();
        foreach event e in events do
            /* For ALLOC, we ignore calls and consider only the
               returns                                              */
            if e.is_call() then
                continue;
            end
            /* A parameter identifier negative indicates no
               concrete value - we ignore it                        */
            if e.pid < 0 then
                continue;
            end
            /* A new address is detected                            */
            if not addr_seen.contains(e.v) then
                /* If the function has less than 3 callers, then
                   we ignore it and do not mark the address as
                   seen                                             */
                if len(callers[e.fid]) <= 3 then
                    continue;
                end
                /* If the function takes an address as parameter,
                   it cannot be ALLOC according to Heuristic 4  */
                if len(P_a[f]) != 0 then
                    continue;
                end
                nb_new_addr[e.fid] ← nb_new_addr[e.fid] + 1;
            end
        end
        /* We return the function f such that nb_new_addr[f] is
           maximum                                                  */
        return nb_new_addr.max_key();
    end
```

Algorithm 21: Get the best candidate for ALLOC

```
Function ComputeAccessors(events, callers, ALLOC)
    Input: events a list of events
    Input: callers a dictionary, where for each function f, callers[f] is
           a list of all the callers of f during the execution
    Input: ALLOC the best candidate for ALLOC
    begin
        accessors ← AddrTableDict();
        /* By default, cells of this dictionary are initialized
           with 0                                                  */
        nb_addr ← Dict();
        /* First step:  we iterate on events and construct the
           list of functions that use each address                 */
        foreach event e in events do
            if e.is_call() then
                if not accessors.contains(e.v) then
                    accessors.add(e.v);
                end
                /* For address e.v, we keep track of the function
                   (and its parameter) that took it as a
                   parameter                                       */
                accessors[e.v].add_accessor(e.fid, e.pid);
            else if e.fid = ALLOC then
                if not accessors.contain(e.v) then
                    accessors.add(e.v);
                end
                /* If a new address was output by ALLOC, we add
                   it to the list of allocated addresses           */
                accessors[e.v].add_accessor(e.fid, 0);
            end
        end
        return accessors;
    end
```

Algorithm 22: Compute the list of accessors for each address

```
Function GetFree(accessors, ALLOC)
    Input: accessors a list of accessors for each address value
    Input: ALLOC the best candidate for ALLOC
    begin
        /* Second step:  we iterate on addresses, and for each
           we get a list of functions that used it (either as a
           return value or as a parameter)                      */
        foreach addr, call in accessors.items() do
            if len(call) = 0 or call.count(ALLOC) = 0 then
                continue;
            end
            /* We split the list of calls with ALLOC: we end up
               with several lists, each one being a sequence of
               calls (ie a couple fid, pid) that used addr      */
            call_seq ← call.split(ALLOC);
            candidates ← List();
            foreach list l in call_seq do
                /* For each list, we keep as a potential
                   candidate the last accessor of the address
                   (before a call to ALLOC or the end of the
                   execution)                                   */
                candidates.add(call_seq.last());
            end
            foreach couple f, p in candidates do
                nb_addr[(f, p)] ← nb_addr + 1;
            end
        end
        /* We return the couple (f, p) corresponding to the
           best candidate for FREE according to the number of
           times it is the last accessor of an address, ie the
           couple (f, p) such that nb_addr[(f, p)] is maximum
           */
        return nb_addr.max_key();
    end
```

Algorithm 23: Get the best candidate for FREE according to a given ALLOC

Chapter 9

Evaluation

This chapter presents experiments, both to assess our approach and to observe the influence of the different parameters. For each type of binary analysis, we propose a metric to assess the accuracy of our results. Regarding arity and types, we propose to compare the results inferred with source-level prototypes of functions on `C` open-source programs. For couples, the metric we use is the number of couples, and the number of unique functions involved in the detected couples. For allocators, we propose to retrieve the well-known standard `libc` allocator in several programs, as well as custom allocators. Finally, we propose a new metric to validate the consistency of a detected allocator which does not require a new execution.

9.1 Methodology

In this section, we present the general methodology we use to validate our implementation for each analysis. In particular, we provide details about the benchmarks we use, the production of oracles and how we produce accuracy results.

9.1.1 Open-source programs compiled from `C`

For all our tests, we use `C` open-source programs, compiled with `gcc`[1] for `Linux x86-64` architecture. Indeed, using open-source programs, compiled from `C`, allows to retrieve easily some source-level information to produce oracles that thus can

[1] to validate *universality*, it would be interesting to test with other compilers as well

be compared with analysis results to evaluate the accuracy. This will be detailed in each of the following sections, but as an example, from the C source-code we can easily retrieve the arity of each function. Note that, because we perform a dynamic instrumentation, we capture functions embedded in the binary under analysis as well as functions from dynamically loaded libraries. Our numeric results include both.

Our benchmark consists of several sets of programs, with different purposes of validation.

9.1.1.1 Detection on small programs

The first benchmark we propose is composed of all 106 programs of the GNU coreutils [osb]. The purpose of this benchmark is to propose tests over a variety of different small programs with different uses and behaviors. We aim to show that even on short executions, our approach allows to retrieve accurate information.

For each program, we provided one input in a configuration file. As an input, we give required parameters for the program to run plus one or two options taken from the manual page. These inputs aim to represent a usual execution of the programs, with no particular thought on the coverage or the time of execution.

As they are small programs, testing all of them is fast, but the coverage is poor (as we perform one single execution), and the number of functions inferred is small (10 to 15 functions for each program).

Note that, in this chapter, each time we present a result with the label coreutils, we actually give the average over all coreutils programs.

9.1.1.2 Influence of inputs

The purpose of this benchmark is to show that the results of our analysis is stable over several executions with different inputs.

We propose three different tests to evaluate this stability:

- 8cc [Uey12] - 8cc is a small C compiler. In this test, we compile with 8cc each of the 43 C source code provided with the compiler on GitHub ([2]).
- ls (from coreutils) - we provide 1000 different sequences of command line arguments for ls. Each sequence, generated randomly from the ls manual page, is composed of five random parameters.

[2] https://github.com/rui314/8cc/tree/master/test

- `bash` [osa] - we provide 17 hand-written bash scripts taken from this bash tutorial: `http://tldp.org/LDP/abs/html/`.

9.1.1.3 Common programs

Finally, we propose a benchmark of several programs that aim to represent the kind of programs we may use every day, either in a personal or in a professional environment.

- `git` [git] - "free and open source distributed version control system".
- `grep` [gre] - "`grep` prints lines that contain a match for a pattern".
- `mupdf-x11` [mup] - "`mupdf` is a lightweight PDF, XPS, and E-book viewer".
- `objdump` [bin] - "`objdump` is a program for displaying various information about object files. For instance, it can be used as a disassembler to view an executable in assembly form. It is part of the GNU Binutils for fine-grained control over executables and other binary data."
- `openssl` [ope] - "OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols".
- `opusenc` [opu] - "open, royalty-free, highly versatile audio codec" that aims to replace `Vorbis` and other proprietary codecs.
- `readelf` [bin] - "Displays information about ELF files". In particular, `readelf` embeds a parser for ELF headers, section table, symbol table, etc.
- `strings` [bin] - "print the strings of printable characters in files."
- `tar` [osc]- "`tar` provides the ability to create tar archives, as well as various other kinds of manipulation".
- `vim` [vim] - "`vim` is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as "vi" with most UNIX systems and with Apple OS X".

9.1.2 Production of an oracle

9.1.2.1 Arity and type

Our tool `scat` embeds a source-code parser to produce oracles for arity and types, based on `clang`. It works in the following way:

- For a given program, we provide to scat a path to the directory containing source files of the program.
- Using `clang`, for each of the files in the given directory, we parse the semantic tree and get every function declaration.
- For each of these functions, we get from its declaration the number of parameters it has.
- For each parameter, we get its undertype by reducing its type to basic types in C[3] (`clang` provides this functionality): if this basic type is `float`, then the oracle's type is `FLOAT`, if the type includes a `*` (dereferencing operator in `C`), then it is `ADDR`, otherwise it is a `NUM`.
- Every undertyped prototype recovered this way is stored to be used later as an oracle.

For all these steps, the corresponding command in `scat` is `parsedata`. `Scat` also handles library dependencies to produce an oracle for dynamically-loaded functions.

9.1.2.2 Allocator

For allocator detection, we produce an oracle manually: for each program of our benchmark, we provide the allocator to be retrieved that we suppose from both source code and assembly (manual) analysis.

9.1.3 Measurements

We have four main points of interest during our experiments, regarding our implementation: the **accuracy** of its results, the **scalability** of our approach, and finally the **influence of parameters and inputs** on the results.

[3] Meaning that every custom type definition is inlined

9.1.3.1 Accuracy

The measurement of accuracy is particularly relevant for the first two analysis we presented: arity and type detection. For each one, we produce an oracle from the source-code and compare directly the results with it. Since the coupling is a new notion we defined, we do not have an external oracle to compare our results with. Finally, regarding allocators, we propose a particular methodology in Section 9.5. In addition, we propose to evaluate the rate of both false positives and false negatives on our results (for each step except couple).

Comparison Every comparison between the results of analysis and the oracle is performed through a `scat` command as well - this is motivated by a fully-automated approach. The corresponding command is `accuracy`. For arity, it compares, for each inferred function, the numeric value of the number of parameters with the oracle, and outputs a percentage of accuracy. For type, it compares for each inferred function, and for each of its parameters, the undertype with the oracle. It also outputs a percentage of correct types.

9.1.3.2 Scalability

To evaluate the scalability of our approach, for each analysis, we present times of execution and overheads. To compute overheads, we make two comparisons:

- `no_pin`: this is the time of execution of the program under analysis when executed in a normal environment (through command line).
- `pin_empty`: this is the time of execution of the program when executed through `Pin`, but with no active nor passive instrumentation - in other words, we measure here the minimum overhead due to `Pin`.

For each analysis, we compute the overhead of our instrumentation relatively to both `no_pin` and `pin_empty`.

9.1.3.3 Influence of parameters

In addition to accuracy, we propose experiments to evaluate the influence of parameters of each analysis on the results. Details are given in each corresponding section, but in a few words, we evaluate the impact of `MIN_CALLS` on both the number of functions inferred and the accuracy of the results for arity analysis. For

types, we evaluate the impact of ADDR_THRESHOLD on accuracy, and the influence of MIN_VALS as well. For coupling, we present the number of couples and left (resp. right) side operands in function of the value of ρ, MIN_VALS and MAX_VALS.

9.1.3.4 Influence of inputs

Because our approach works on a single trace of execution for each step, we must ensure that the results are not too much dependent on this particular trace, and so on the inputs given to a program. To evaluate this dependency, we perform experiments on 8cc, ls and bash, over various inputs, to test the influence of the trace of execution on the results. This experiment is to show that our results are stable regarding the input (in terms of accuracy and scalability).

9.1.4 Reproductibility of experiments

To make our tests reproducible, they are fully automated and integrated to scat, as well as the tested programs and inputs. Everything needed to re-run our tests is available in the GitHub repository of scat. Note that it includes the production of latex tabulars and charts as well as the actual execution of tests. For each result we present, we provide the scat command line used to run the tests.

9.1.5 Platform

We run our tests on a Linux Mint 17 64 bits virtualized with Virtual Box 4.3.20. The host characteristics are an Intel Core i7-4610M and 16 Go of RAM. The virtual machine was attributed two CPU cores and 8GB of RAM. We use Pin 2.14 for the instrumentation and CLang 1-3.4 to produce oracles from source.

9.2 Arity

In this section, we present the results of our tests relatively to arity.

9.2.1 Metric

For each of our tests, we compare the results of our inference with the data recovered from the source code. Because of dynamic linking and imperfections of our implementation, we do not recover, from source, every function that is

embedded in the binary. Therefore, there are some functions that we do detect but for which we do not have an oracle. These functions are excluded from our statistics.

Accuracy The accuracy we present is the percentage of functions for which the arity of parameters was correctly inferred. An accuracy of 90% means that, among the functions we inferred and for which we have an oracle, the result is correct for 90% of them. The same computation stands for the accuracy of the arity relative to return parameters. Note, however, that the arity of return values, in our context, is necessarily 0 (function not returning any value) or 1 (function returning either an integer or a float value).

More precisely, given the set of functions inferred for which we have an oracle denoted by $\mathcal{F}$, for each function $f \in \mathcal{F}$, we denote by $\#_p f$ and $\#_r f$ the arity (resp. for parameters and return value) of f according to the oracle, and $\#_p^i f$ and $\#_r^i f$ the result of the arity analysis of f. Then, the accuracy is computed as follows (for return parameters, replace $\#_p^i f$ by $\#_r^i f$ and $\#_p f$ by $\#_r f$):

$$\text{accuracy} = \frac{|\{f \in \mathcal{F} \mid \#_p^i f = \#_p f\}|}{|\mathcal{F}|}$$

False positives and false negatives

For parameters Among the errors we get from the detection, sometimes we detect too many parameters, and sometimes we miss parameters. The rate of false positives, denoted by e_{fp}, is the rate of errors due to a detected arity higher than expected, and the rate of false negatives, denoted by e_{fn}, is the rate of errors due to a detected arity less than expected.

$$e_{fp} = \frac{|\{f \in \mathcal{F} \mid \#_p^i f > \#_p f\}|}{|\mathcal{F}|}$$

$$e_{fn} = \frac{|\{f \in \mathcal{F} \mid \#_p^i f < \#_p f\}|}{|\mathcal{F}|}$$

For return values In the same way, we define false positives and false negatives over the return values as follows:

$$e_{fp} = \frac{|\{f \in \mathcal{F} \mid \#_r^i f > \#_r f\}|}{|\mathcal{F}|}$$

$$e_{fn} = \frac{|\{f \in \mathcal{F} \mid \#_r^i f < \#_r f\}|}{|\mathcal{F}|}$$

Number of functions tested For each test, we also present the number of functions tested, *i.e.* the total number of functions inferred minus the functions for which we do not have any oracle: $|\mathcal{F}|$.

9.2.2 Results

Our results have been produced using `scat`: tests have been fully automatized and are reproducible (see `test` and `chart` commands in the documentation of `scat` - `https://github.com/Frky/scat`).

9.2.2.1 General results

```
scat > test arity accuracy -t test/config/general.yaml
scat > chart arity accuracy general
```

Table 9.1 presents the results of our arity analysis, for every entry of our benchmark. These results have been obtained with the following values for parameters: `MIN_CALLS = 5`, `PARAM_THRESHOLD = RET_THRESHOLD = 0.1`. Particular entries `coreutils` and `8cc` are an average of, respectively, every program in the `coreutils` and every trace of `8cc` (with different inputs) - therefore, there are overlaps and the total number of functions tested may include several times the same functions.

Discussion First, these results show that our approach is accurate: we achieve an accuracy rate of 93% for the arity of parameters, and 93% for return values, on a total of 2000 functions. It also shows that the number of false positives, is lower than the number of false negatives. In particular, we end up with zero false positive in the detection of return values.

	accuracy (in %)		false neg.		false pos.		total tested	
	param	ret	param	ret	param	ret	param	ret
8cc	0.98	0.97	4	8	2	0	283	283
bash	0.95	0.95	8	15	7	0	283	283
coreutils	0.93	0.82	154	448	15	0	2515	2515
git	0.96	0.93	15	33	6	0	492	492
grep	0.91	0.84	6	14	2	0	86	86
mupdf-x11	0.94	0.95	13	17	7	0	348	348
objdump	0.89	0.96	7	5	7	0	132	132
openssl	0.96	0.96	4	8	3	0	194	194
opusenc	0.94	0.86	2	5	0	0	36	36
readelf	0.93	0.95	4	3	0	0	59	59
strings	0.95	0.90	1	2	0	0	20	20
tar	0.85	0.86	16	15	0	0	106	106
vim	0.91	0.90	18	27	8	0	275	275
TOTAL	0.93	0.93	94	144	40	0	2031	2031

Table 9.1: General results of arity detection in one execution on `coreutils`, `8cc` and common applications

False negatives False negatives are mostly due to unused parameters. Sometimes, it is because the parameter is obsolete (and has been kept for compatibility reasons), and sometimes because it was not used in our trace of execution because we did not activate all paths. Regarding return values, most of false negatives are due to a return value that is not checked (`e.g.` the return value of `printf`).

False positives False positives are mostly due to an improper implementation. Missing some instructions performing a write in a register may lead to errors of this kind. Indeed, we observe a *read-before-write* event on a given register and then deduce a parameter whereas the memory location has been written since last call.

9.2.2.2 Influence of parameters

```
scat > test arity <param_to_test> -t test/config/8cc.yaml
scat > chart arity <param_to_draw> 8cc
```

To assess the influence of each parameter of the inference, we ran our tests on `8cc` with different values for each. Figure 9.1 presents these results. For each

of the sub-figures, one parameter varies while the others remain at their default values: `MIN_CALLS = 10`, `PARAM_THRESHOLD = RET_THRESHOLD = 0.1`. These measurements were performed on the `8cc` program with 43 different inputs. Each point corresponds to an average over all inputs for a given set of values for parameters.

Discussion These results allow to conclude on the interest of each parameter of the analysis.

`MIN_CALLS` Figure 9.1a illustrates two points. First, as expected, the value of `MIN_CALLS` impacts the number of functions we can infer: the higher the value, the less functions are executed at least `MIN_CALLS` times during one execution. Second, its value impact, but not significantly, the results of accuracy. We performed the same tests on `coreutils` and on our common programs, and got similar results. We make two hypothesis to explain this:

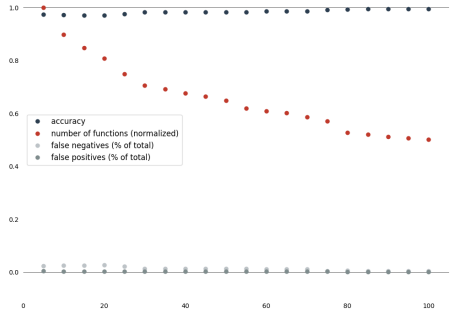
`PARAM_THRESHOLD` The influence of `PARAM_THRESHOLD` is shown in Figure 9.1b. We observe that, for `PARAM_THRESHOLD < 0.4`, this parameter has no influence on the accuracy of our results. However, from `0.4`, it has a negative impact on the accuracy, as we miss more and more parameters. In addition, the rate of false positives is not significantly impacted by the value of the parameter.

`RET_THRESHOLD` Figure 9.1c shows explicitly that this parameter has no influence at all on the results of arity detection. We explain that by the fact that relevant return values of functions are always fetched, whereas non-import ones are almost always ignored.

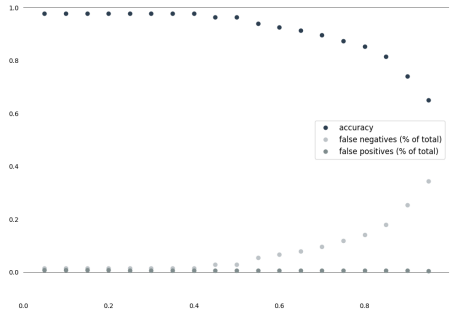
9.2.2.3 Influence of the input

```
scat > test arity accuracy -t test/config/8cc.yaml
scat > chart arity variability 8cc
```

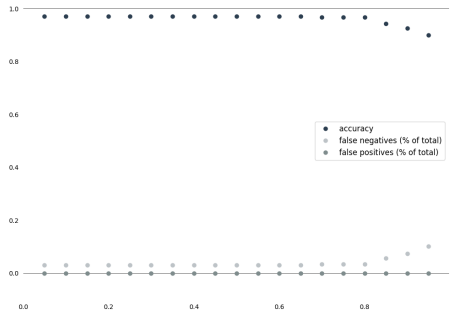
Figure 9.2 presents the accuracy of the arity analysis for every entry (*i.e.* for every source code that we compile) of `8cc` with the following values of parameters: `MIN_CALLS = 10`, `PARAM_THRESHOLD = RET_THRESHOLD = 0.1`.



(a) `MIN_CALLS` from 5 to 100



(b) `PARAM_THRESHOLD` from `0.05` to `1`



(c) `RET_THRESHOLD` from `0.05` to `1`

Figure 9.1: Influence of parameters on arity detection over 8cc

Each bar corresponds to one particular input of `8cc`. The number on the x-axis corresponds to the number of functions inferred during the analysis. On the y-axis, we have first the accuracy (in %), and then the percentage of false negatives and positives. The total (accuracy (in %) + false positives (in %) + false negative (in %)) is logically equal to 1. Please note that the y-axis starts from `0.90`.

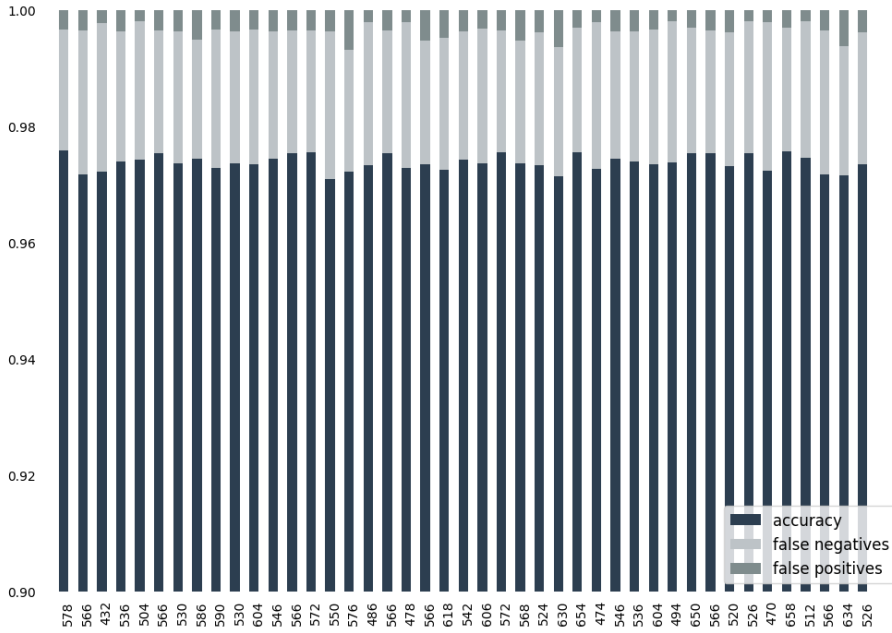


Figure 9.2: Influence of the input on the accuracy of arity inference - tested on `8cc` compiler

In addition, we performed the same tests on `ls` and `bash`. For these three sets of tests, we present in Listing 9.1 the average and standard error of the accuracy, the false positive and the false negative rates. These measures confirm that the results of analysis are very stable regarding the inputs.

```
[8cc]
average/standard deviation:
```

```
| accuracy: 0.974/0.0013
| false positive: 0.00358/0.00117
| false negative: 0.0227/0.00147
[bash]
average/standard deviation:
| accuracy: 0.943/0.0081
| false positive: 0.0103/0.00428
| false negative: 0.0462/0.0102
[ls]
average/standard deviation:
| accuracy: 0.908/0.0224
| false positive: 0.00288/0.00489
| false negative: 0.0894/0.02
```

Listing 9.1: Average and standard error on accuracy, false positives and false negatives on arity detection when the input varies

Discussion Working on a single execution could lead to results that are highly dependent on the trace we observe. Figure 9.2 shows that, in facts, our arity detection is stable over inputs: with 43 different inputs, we end up with different functions inferred, but the accuracy is very stable as well as the number of false positives and negatives.

9.2.3 Overhead

Table 9.2 presents the overhead of our instrumentation, relatively to a normal execution (`no_pin`) and to an execution using `Pin` but performing no instrumentation. These experiments show two important points. First, the overhead in percentage is significant (up to 51589% on `objdump`). Although the major part of this overhead is due to `Pin`, it is still to be endured. Second, and despite this high overhead, the actual times of execution remain acceptable: no analysis lasts longer than fifteen seconds, whereas we target programs whose sizes go up to 13MB.

These timing results are still to be discussed. Indeed, some of these programs have a lot of i/o (in particular `tar`), and these i/o can delay the execution (and thus attenuate the impact of the instrumentation). In addition, `Pin` induces a cost of instrumentation, before the program starts the actual execution, that does not depend on the time of execution. Therefore, on small programs (such as the `coreutils`), this fixed cost increases the overhead more importantly than with larger programs (*e.g.*, `vim`).

program	size	no_pin	empty		online	
`8cc`	293 KB	0.019	0.668	3481%	3.629	18903%
`bash`	1 MB	0.064	4.207	6565%	8.089	12624%
`coreutils`	132 KB	0.004	0.235	6014%	0.862	22057%
`git`	5 MB	0.694	2.259	325%	5.344	770%
`grep`	687 KB	0.006	0.505	7958%	1.748	27573%
`mupdf-x11`	13 MB	2.189	3.262	149%	14.856	678%
`objdump`	2 MB	0.005	0.586	12596%	2.400	51589%
`openssl`	778 KB	0.010	0.692	7251%	2.102	22010%
`opusenc`	111 KB	0.111	0.824	740%	9.042	8124%
`readelf`	660 KB	0.005	0.554	10163%	1.974	36245%
`strings`	1 MB	0.003	0.284	8255%	0.866	25206%
`tar`	1 MB	5.812	5.829	100%	7.181	123%
`vim`	8 MB	0.838	4.997	596%	7.877	940%

Table 9.2: Overhead of the arity detection on `8cc`, `coreutils` and common programs

9.3 Type

The experiments for type accuracy are very similar to the ones for arity.

9.3.1 Metric

Regarding types, we use the same metrics as the ones presented in previous section for arity.

We denote by P the set of parameters such that:

1. $p \in P$ is a parameter of a function f for which the arity was correctly inferred at previous step,
2. we do have an oracle for the type of p, that we denote by $type(p)$,
3. we have inferred a type, denoted by $type^i(p)$.

9.3.1.1 Accuracy

The accuracy we compute for type is the following:

$$\text{accuracy} = \frac{|\{p \in P \mid \text{type}(p) = \text{type}^i(p)\}|}{|P|}$$

In other words, the accuracy is the number of parameters we correctly infer, considering only parameters of functions for which we correctly inferred the arity, and such that we have an oracle for the type.

9.3.1.2 False positives and false negatives

According to our definitions, a false positive is a parameter detected as an address whereas it is not, and vice-versa for a false negative.

$$e_{fp} = \frac{|\{p \in P \mid \text{type}(p) \neq \text{type}^i(p) \text{ and } \text{type}^i(p) = \texttt{ADDR}\}|}{|P|}$$

$$e_{fn} = \frac{|\{p \in P \mid \text{type}(p) \neq \text{type}^i(p) \text{ and } \text{type}(p) = \texttt{ADDR}\}|}{|P|}$$

9.3.2 Results

As for arity, we present three kind of results here: general results (see Section 9.3.2.1), influence of parameters on the results (see section 9.3.2.2) and influence of inputs on the results (see Section 9.3.2.3).

9.3.2.1 General results

```
scat > test type accuracy -t test/config/general.yaml
scat > chart type accuracy general
```

Table 9.3 shows the accuracy of type analysis for every entry of our benchmark. These results have been obtained with the following values for parameters: `MIN_VALS = 10` and `ADDR_THRESHOLD = 0.5`. Note that the accuracy is computed for every parameter that is both detected by the arity step and in the oracle: it means that a parameter detected by the arity step that does not exist (according to the oracle) is not taken into account ; the same stands for parameters that are in the oracle but not detected.

Discussion As for arity, these results show that our implementation is accurate: on average, the type of parameters is successfully detected in 96% of the cases. The accuracy for return parameters is lower, but we have no explanation for the difference.

	accuracy (in %)		false neg.		false pos.		total tested	
	param	ret	param	ret	param	ret	param	ret
8cc	0.98	0.96	4	7	3	2	307	216
bash	0.96	0.90	14	20	2	4	374	239
coreutils	0.92	0.81	208	276	55	7	3299	1528
git	0.94	0.92	26	7	4	13	530	263
grep	0.95	0.88	4	7	3	0	129	58
mupdf-x11	0.96	0.93	17	10	11	12	746	305
objdump	0.94	0.92	3	4	12	4	231	105
openssl	0.95	0.85	9	13	5	11	308	160
opusenc	0.98	1.00	0	0	1	0	53	24
readelf	0.94	0.91	1	3	5	1	106	47
strings	0.94	0.82	2	3	0	0	34	17
tar	0.96	0.91	4	6	1	1	142	75
vim	0.97	0.93	8	11	2	2	342	176
TOTAL	0.96	0.91	88	84	46	48	2995	1469

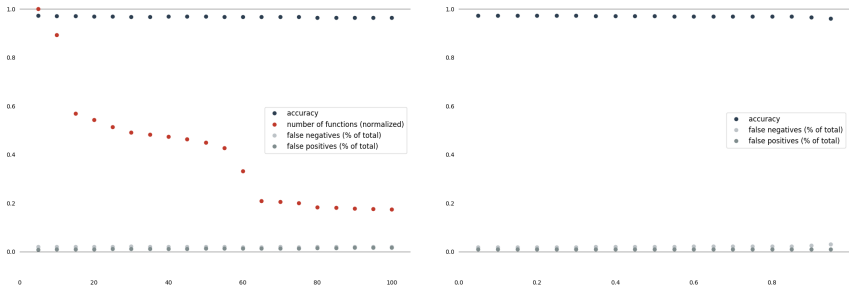
Table 9.3: General results of type detection in one execution on `coreutils`, `8cc` and common applications

9.3.2.2 Influence of parameters

```
scat > test type <param_to_test> -t test/config/8cc.yaml
scat > chart type <param_to_draw> 8cc
```

To evaluate the influence of each parameter of the inference, we ran our tests on `8cc` with different values of each. For a given value, we run 43 times `8cc`, with 43 different inputs. Figure 9.3 presents these results. For each of the sub-figures, one parameter varies while the other remains at its default value: `MIN_VALS = 10`, `ADDR_THRESHOLD = 0.5`. In Figure 9.3a, `MIN_VALS` varies from 5 to 200 with a step of 5. For each point, we compute the average (of accuracy, number of functions, etc.) on each execution of `8cc` (corresponding to one input). In Figure 9.3b, `ADDR_THRESHOLD` varies from `0.05` to `1` with a step of `0.05`.

Discussion From these experiments, we can conclude on the impact of each parameter on the results.



(a) `MIN_VALS` from `5` to `100` (b) `ADDR_THRESHOLD` from `0.05` to `1`

Figure 9.3: Influence of parameters on type detection over `8cc`

`MIN_VALS` Figure 9.3a shows that the `MIN_VALS` parameter has a significant impact on the number of functions inferred, which is logical: it has the same role as `MIN_CALLS` had for arity detection. Therefore, the more we require values to conclude, the more we need the function to be executed, and thus the less functions are to be inferred.

This experiment shows that, for type inference, this parameter has almost no impact on accuracy. This is because the accuracy of the type inference relies on the inference of the addressable space rather than on function calls. We conclude from this that types of parameters can be inferred over a very small number of concrete values.

`ADDR_THRESHOLD` In Figure 9.3b, we observe an interesting phenomenon: `ADDR_THRESHOLD` has almost no influence on the accuracy of the type inference. From `ADDR_THRESHOLD = 0.80`, we start observing an increase of the number of false negatives, and then the total accuracy, but this variation is not significant. We explain this fact by the method we use to detect addresses. First, we infer the whole address space, and then every concrete value that is between the bounds of this address space is considered as an address. Because we do not perform a particular detection (through data flow for instance) for each value of parameter (we only check its membership to the address space inferred), there is no reason we would have a different behavior from one execution of a function to another. These results confirm this hypothesis.

9.3.2.3 Influence of the input

```
scat > test type variability -t test/config/8cc.yaml scat
> chart type variability 8cc
```

Here again, we test this influence by performing type inference on the `C` compiler `8cc` on the 43 different inputs, with the default values for each parameter (`MIN_VALS = 10`, `ADDR_THRESHOLD = 0.5`). For each, we present the accuracy in percentage, as well as false positives and false negatives - see Figure 9.4. Please note that the y-axis starts from `0.90`. As for arity, the numbers below each bar corresponds to the number of parameters and return values tested.

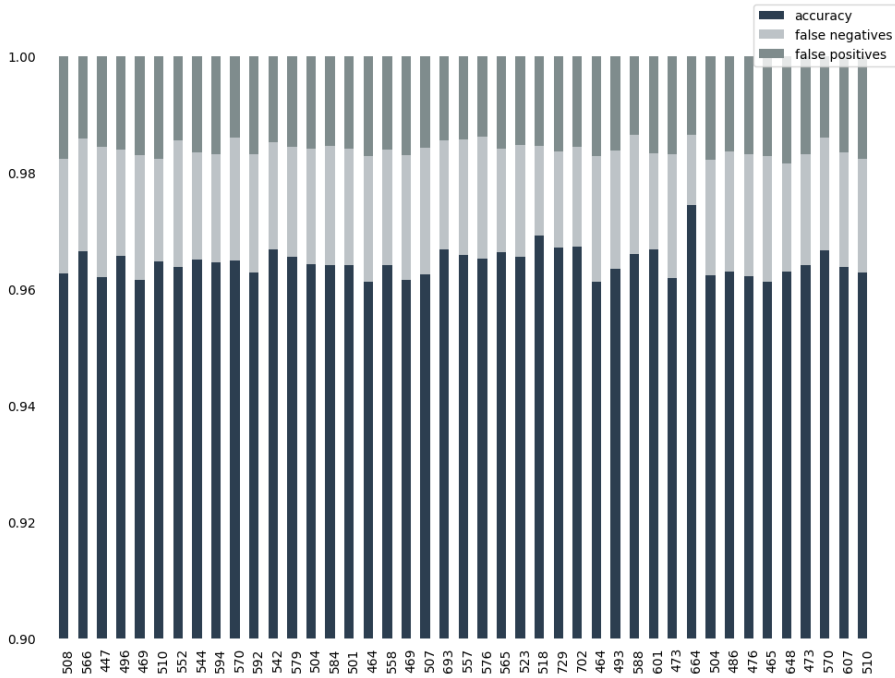


Figure 9.4: Influence of the input on the accuracy of type inference - tested on `8cc` compiler

As for arity, we also performed the same experiment on `ls` (with various

command line arguments) and on `bash` (with various scripts to execute). Listing 9.2 presents the average and standard error for each.

```
[8cc]
average/standard deviation:
| accuracy: 0.97/0.00265
| false positive: 0.00964/0.00113
| false negative: 0.0207/0.00202
[ls]
average/standard deviation:
| accuracy: 0.888/0.0501
| false positive: 0.00792/0.00907
| false negative: 0.105/0.0561
[bash]
average/standard deviation:
| accuracy: 0.957/0.0104
| false positive: 0.0073/0.00154
| false negative: 0.0359/0.00962
```

Listing 9.2: Average and standard error on accuracy, false positives and false negatives of type detection when the input varies

Discussion On this set of data, we observe that the influence of the input is very small: the lower accuracy we get is around `0.96` and the higher is below `0.98`. The number of false positives and false negatives is also stable in function of the input (around `0.015` for each). This is confirmed by the low standard error on `ls` and `bash`.

9.3.3 Overhead

In Table 9.4, we present time measurements and overhead of our instrumentation, relatively to both a non-instrumented execution and an execution through `Pin` with no runtime inspection. These results lead to the same conclusion as for arity: the overheads very are similar.

9.4 Couple

This section presents our results on couple detection. Whereas, for arity and type, we had an easy way to compute accuracy by comparison of the inferred results with an oracle, such oracle does not exist for coupling. The metrics we propose

program	size	no_pin	empty		online	
`8cc`	293 KB	0.019	0.668	3481%	4.253	22153%
`bash`	1 MB	0.064	4.207	6565%	7.356	11480%
`coreutils`	132 KB	0.004	0.235	6014%	0.590	15116%
`git`	5 MB	0.694	2.259	325%	4.780	688%
`grep`	687 KB	0.006	0.505	7958%	1.510	23808%
`mupdf-x11`	13 MB	2.189	3.262	149%	15.576	711%
`objdump`	2 MB	0.005	0.586	12596%	2.103	45186%
`openssl`	778 KB	0.010	0.692	7251%	2.067	21649%
`opusenc`	111 KB	0.111	0.824	740%	4.006	3599%
`readelf`	660 KB	0.005	0.554	10163%	1.904	34951%
`strings`	1 MB	0.003	0.284	8255%	0.799	23248%
`tar`	1 MB	5.812	5.829	100%	7.982	137%
`vim`	8 MB	0.838	4.997	596%	7.938	947%

Table 9.4: Overhead of the type detection on `8cc`, `coreutils` and common programs

are then different. Note that a second evaluation of couple is presented in Section 9.5: indeed, coupling helps detecting allocators efficiently.

9.4.1 Metric

The indicators we present in this section are the following: `#function` the total number of functions candidates for couples (*i.e.*, functions with at least one address in their prototype), `#couples` the number of couples detected, `#left` and `#right` the number of left (resp. right) operands in couples, *i.e.* the number of different functions f (resp. g) such that there exists at least one couple (f, g) inferred.

Despite the lack of oracle, as the notion of coupling is a notion we introduce in this work, our experiments aim to show that:

- relatively to the number of candidates, the number of couples is much lower than the number of possible combinations; this aims to emphasize the relevance of the notion of coupling as a way to identify data flows,

- relatively to the number of couples, the number of left and right operands is low; in particular, a few functions are involved in couples as a left operand. This means that there are few sources of addresses, and many consumers.

9.4.2 Results

In the following sections, we present general results with these indicators - Section 9.4.2.1, the influence of parameters - Section 9.4.2.2, and the influence of inputs - Section 9.4.2.3.

9.4.2.1 General results

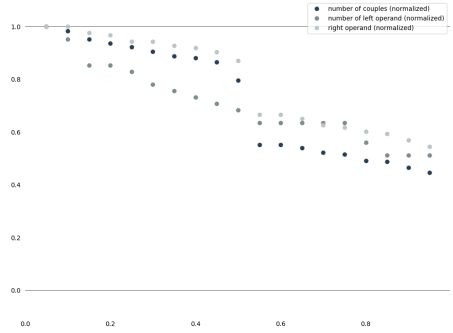
```
scat > test couple general -t test/test/general.yaml
scat > chart couple general general
```

Table 9.5 presents the general results of the couple detection. In this table, we observe, first, that the number of couples is generally lower that the number of instrumented functions (*i.e.* functions dealing with addresses). This shows that the notion of coupling has an interest: we do not end up with all the possible couples (which would be the number of functions to the power two). Second, we can note that the number of left-side operand in couples is significantly lower than the number of right-side operands. This is an interesting point, if we anticipate on allocator detection: it means that sources of addresses in coupling are not many. In other words, a few different functions seem to be sources of couples, regarding the total number of couples.

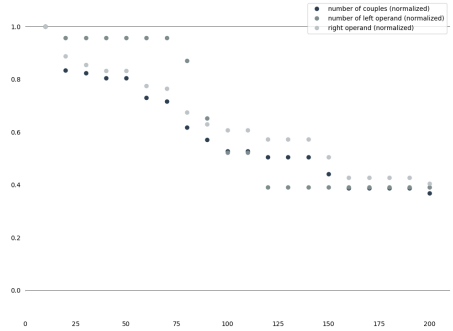
9.4.2.2 Influence of parameters

```
scat > test couple <param_to_test> -t test/config/8cc.yaml
scat > chart couple <param_to_draw> 8cc
```

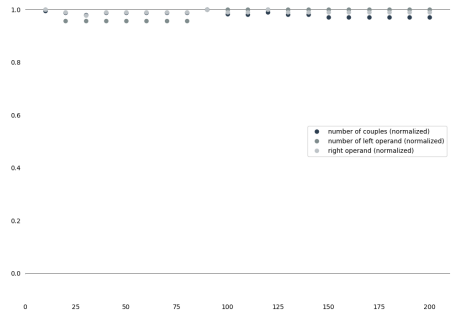
As a reminder, couple inference has three parameters: ρ, which is the threshold for which we want to find couples, `MIN_VALS` and `MAX_VALS` that are parameters relative to the number of values we should get to conclude on coupling. Results presented in Figure 9.5 show the influence of these parameters on the indicators we mentioned earlier, tested on `8cc`. In each subfigure, we normalized the data: we are interested in the trend rather than in the numeric values. For each, one parameter varies while the others take their default value: $\rho = 0.5$, `MIN_VALS` = 10 and `MAX_VALS` = 200. In Figure 9.5a, ρ varies from `0.05` to `1` with a step of `0.05`. In Figure 9.5b, `MIN_VALS` varies from 10 to 200 with a step of 10, and Figure 9.5c is the same for `MAX_VALS`.



(a) Influence of the parameter `rho` on the number of couples and left/side operands - tested on `8cc`



(b) Influence of the parameter `MIN_VALS` on the number of couples and left/side operands - tested on `8cc`



(c) Influence of the parameter `MAX_VALS` on the number of couples and left/side operands - tested on `8cc`

Figure 9.5: Influence of parameters on couple detection over `coreutils`

	functions	couples	left	right	online (s)	offline (s)
8cc	338	465	75	97	2.61	18.85
bash	380	241	42	42	4.52	7.24
coreutils	54	12	3	3	0.36	0.72
git	653	16	4	4	0.68	3.39
grep	156	47	12	16	0.97	2.74
mupdf-x11	1105	1750	97	335	10.54	54.73
objdump	187	48	8	17	1.30	3.68
openssl	231	370	24	76	1.41	5.11
opusenc	195	33	9	16	2.10	4.57
readelf	109	39	10	15	1.11	2.78
strings	55	6	3	2	0.44	0.90
tar	182	73	19	24	6.95	4.31
vim	410	161	23	45	50.60	6.87
TOTAL	4055	3261	329	692	2646.39	115.89

Table 9.5: General results of couple detection in one execution on `coreutils`, `8cc` and common applications

Discussion

ρ The parameter ρ is the most important parameter of this step, because it directly influences the couples we want to retrieve. Without surprise, the higher ρ is, the less couples are detected. Between ρ = 0.05 and ρ = 1, the number of couples detected is divided by two. Number of left and right operands are impacted in similar ways. However, we observe that there is a clear discontinuity around $\rho = 0.5$ for the number of couples and the number of right operands, whereas the number of left operands seems not to be subject to it. We have no clear explanation of this phenomenon yet: a manual analysis would be required to understand what is happening with $\rho = 0.5$.

`MIN_VALS` The `MIN_VALS` parameter influences the number of couples we get from the analysis. We distinguish three phases in the evolution; from `MIN_VALS` = 10 to `MIN_VALS` = 70, the number of couples lowers, and the number of right operands too. However, the number of left operands is not impacted. Then the number of left operands drastically decreases: it is almost divided by two from `MIN_VALS` = 70 to `MIN_VALS` = 100. Then, from 110 to 140, the three indicators are stable: this is the second phase. During the third phase, *i.e.* from 150 to 200,

we observe a slow diminution of the number of couples and the number of right operands, but once again the number of left operands is stable.

From this, we deduce that, except for points of discontinuity, the number of left operands is not impacted by the value of MIN_VALS, whereas the number of couples and the number of right operands are. This is because, for the left operands, we keep every single value that is output, independently from MIN_VALS (which only state the number of values of parameters we need to conclude). In conclusion, it is possible to adjust the value of MIN_VALS depending on what we want: if we want a high number of left operands, we should take a low value, and vice-versa. If we want as much couples as possible, we should take a value of MIN_VALS at the beginning of a phase (depending on what we want regarding the number of left operands), and otherwise we should favor a value at the end of a phase.

MAX_VALS This parameter, according to Figure 9.5c, has no impact on the number of couples, nor on the number of left/right operands. This seems to indicate that we can perform coupling statistics on a few number of values: from 100 values, there is no need to collect more as it does not impact the results.

9.4.2.3 Influence of the input

```
scat > chart couple variability
```

We propose to observe the impact of the inputs on the number of couples we infer. Once again, we produce our data from the C compiler 8cc, with 43 different inputs, and each time the same values of parameters (the default ones). Figure 9.6 presents the results. In this figure, each bar corresponds to one input of 8cc. For each, we present the number of couples, the number of left operands and the number of right operands.

Discussion This time, we observe that the input has an impact on the couples we detect for ρ = 0.5: the number of couples varies from around 400 to more than 700. However, half of the inputs lead to the detection of 400 couples, $\pm$ 5%.

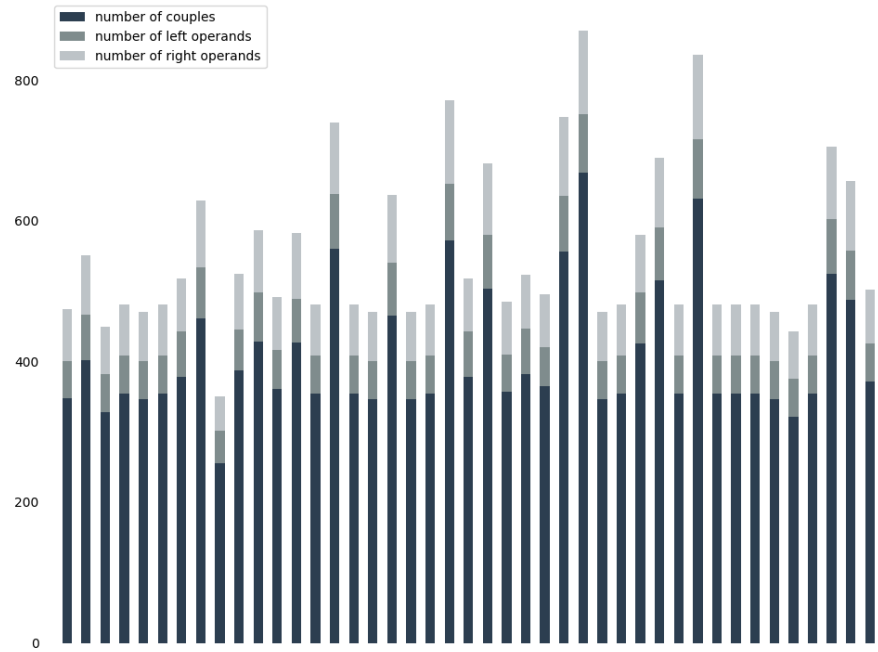


Figure 9.6: Influence of the input on the number of couples and left/side operands - tested on `8cc` compiler

9.5 Allocators

This section presents our experiments on allocators. To our knowledge, there is no free benchmark available relatively to allocator detection. This makes the evaluation of our implementation hard to automatize. We propose a benchmark composed of several programs using `malloc` and `free` from the `libc`, that we try to detect. For these, the oracle is easy to construct, and tests can be performed on a reasonable amount of programs with very little manual intervention. However, the obvious drawback is that it does not evaluate the capacity of our approach to detect other allocators. We also provide several examples of custom allocators we successfully detect with `scat`, but these examples are very long to construct: we need to find open-source programs that do not use the `libc` standard allocator,

and then find the custom allocator to construct an oracle. For this reason, we do not provide a large number of examples with custom allocators.

9.5.1 Metric

To evaluate our approach, we propose several metrics and experiments.

9.5.1.1 Do we retrieve allocators?

First, we propose to test the allocator detection with a "yes/no" oracle (manually provided): these tests aim to show the capability of `scat` to successfully retrieve allocators in binaries.

9.5.1.2 How to evaluate the consistency of our results?

To evaluate the consistency of a supposed allocator (`ALLOC`, `FREE`), we propose to check two main properties that it should satisfy. For each property, we count an error each time it is violated.

Errors First, a `FREE` should only occur on addresses allocated by `ALLOC`. This means that every time `FREE` is called with a parameter that was not output by `ALLOC` before, we detect an error. Second, an `ALLOC` should not return the same address twice if it was not released by `FREE` in between. Here again, every time we see an *allocated* address being re-allocated, we count an error.

Error rate We compute the error rate as the total number of errors (generated by either `ALLOC` or `FREE`) divided by the number of calls to `ALLOC` plus the number of calls to `FREE`. If this error rate is below a threshold, we validate the detected allocator. Otherwise, we conclude that the couple (`ALLOC`, `FREE`) is inconsistent.

Note that, among the two properties we presented, one is relative to `ALLOC` and the other is relative to `FREE`. However, if one or the other of these functions was inferred incorrectly, it would most certainly lead to errors relative to both. For instance, a misdetection of `FREE` would lead `ALLOC` to generate errors, because blocks would not be freed properly (considering the `FREE` function that was inferred). That is the reason why we only consider the global error rate rather than two error rates (one for `ALLOC` and one for `FREE`), and we use it to validate or invalidate the

allocator **as a couple**, but one of the two functions can still be correct, and we could use other indicators to invalidate each one individually.

Consistency We present in Section 9.5.2.2 an experiment to show that the consistency rate, *i.e.* one minus the error rate, is very high for allocators, and that it is significantly lower for other couples. We set our threshold at `0.95` (*i.e.* a couple with a consistency rate above `0.95` should be an allocator, whereas a couple with a lower consistency rate should not be an allocator), and our experiment show that it is indeed a good value.

Cost Note that the consistency rate can be computed on the same data as we compute `ALLOC` and `FREE`: therefore it does not require any new execution.

9.5.2 Results

9.5.2.1 General results

```
scat > test alloc couple -t test/config/general.yaml
scat > chart alloc couple -t test/config/general.yaml
```

Tables 9.6 and 9.7 presents the general results of our allocator detection, based on the results of the coupling inference, respectively on `coreutils` programs and other programs from our test suit. Each of these programs use the standard `libc` allocator. In both tables, the second column shows if the functions `ALLOC`/`FREE` where correctly inferred (✓) or not (×) - *n.c.* indicates that no function was output as a good candidate. The third column presents the error rate (*i.e.* one minus the consistency rate). The third and fourth columns present the time of execution in seconds for, respectively, the online step and the two offline steps (one to retrieve `ALLOC` and one to retrieve `FREE`).

Discussion First, we observe that every time our analysis outputs an `ALLOC`, it is the correct one. No candidate is output at all (*n.c.*) in 4 cases. On the other hand, `FREE` is retrieved in 26 over 48 times, which is less good - in 8 cases, no candidate for `FREE` is output at all, *i.e.* 4 times when `ALLOC` was detected. Note that, in facts, no allocator was found in much more cases: we did not include, in Table 9.6, programs in `coreutils` for which no candidate for `ALLOC` was output.

It is not rare on small programs, because the trace is often too short to be able to differentiate the allocator from other couples. This also means that Table 9.6 presents every single result on `coreutils` for which we output an `ALLOC` function: there is no hidden results here.

Second, errors on the detection of `FREE` are almost always detected by an error rate above the threshold of `0.05`. Actually, we present in Section 9.5.2.2 experiments to show that this error rate is a very good criterion to detect errors in the allocator detection.

Finally, we can observe that the time of execution is very acceptable: around one second for `coreutils` programs. In addition, the offline computation is also quick enough to be scalable.

9.5.2.2 Consistency rate

```
scat > test alloc consistency -t test/config/general.yaml
scat > chart alloc consistency -t test/config/general.yaml
```

To evaluate the relevance of the consistency rate, we propose the following experiment: for each program of our benchmark, we test every candidate for `ALLOC`, and for each of them, we test every couple (`ALLOC`, `FREE`) for the 3 best candidates for `FREE` relatively to `ALLOC`. For each couple, we compute the consistency rate, and see if it is compatible with the correctness of the couple:

- for the good couple (`ALLOC`, `FREE`) (*i.e.* the allocator we are looking for),
  - if the consistency rate is above `0.95`, then it is compliant (dark-blue dot);
  - if the consistency rate is below `0.95`, then we have a false negative (light-gray dot);
- for wrong couples,
  - if the consistency rate is above `0.95`, then we have a false positive (red dot if both `ALLOC` and `FREE` are incorrect, orange dot if `ALLOC` is correct);
  - otherwise it is compliant (dark-blue dot).

Figure 9.7 presents the results of this experiment. The horizontal line shows the `0.95` threshold, and vertical lines separate the different programs of our benchmark.

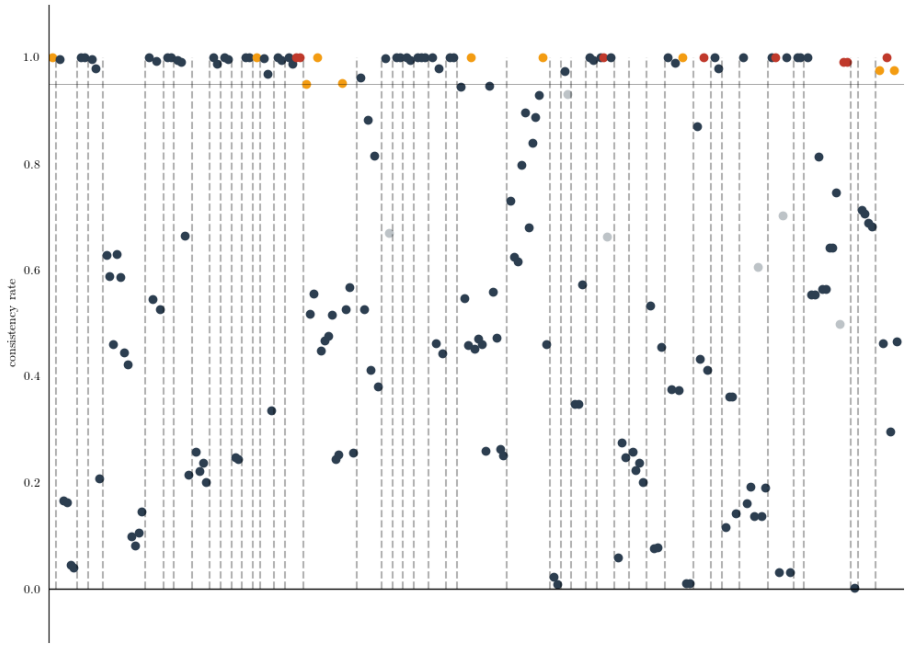


Figure 9.7: Relevancy of the consistency rate to detect wrong allocators

Discussion At first look, this experiment shows that the consistency rate is a good indicator: a large majority of dots are dark-blue, which means that the rate is consistent with the correctness of the allocator. However, it presents false negatives and false positives that we try to explain in this section.

The red dots: false positives False positives are the worst type of error we can get: it means that the consistency rate validates a couple whereas both `ALLOC` and `FREE` were wrongly inferred. It occurs 8 times. We manually analyzed these eight cases, and we found that:

- In two cases,we end up with an alternative allocator: the program does not use a unique allocator during the execution, but there is a unique `FREE` function. Our heuristics are not suited to detect such cases.
- In three cases, we do not find memory allocators but functions that could corre-

spond, in a certain way, to the definition of allocators of other types. In particular, we have functions of IO on files (*e.g.* (`find_line`, `save_line_to_file`)) that could be seen as allocators of lines. Another example is the couple (`new_fd_bitmap`, `dispose_fd_bitmap`) which is an allocator of bitmaps.

- Three cases would require a deeper manual investigation.

The orange dots: half-false positives In 9 cases, the consistency rate validates a couple (`ALLOC`, `FREE`) whereas only `ALLOC` is correct. In these cases, a very few errors are detected because, although `FREE` is wrongly inferred, a very few reallocations occur. Since reallocation errors are the only ones allowing to detect a wrong `FREE` function, we obtain false positives in these cases.

The light-gray dots: false negatives We have 6 false negatives, *i.e.* couples that correspond to the correct allocator but that are rejected by the consistency rate criterion.

9.5.2.3 Custom allocators

Manual and early experiments on several programs show that our method is able to find custom allocators.

`Jasper` `Scat` is able to retrieve the custom allocator embedded in `jasper`, Using couples, we detect the custom allocator embedded in `jasper`: (`jas_malloc`, `jas_free`). The computation of the consistency gives the following results: we end up with 6 errors in 204 calls, that is an error rate of 2.94×10^{-2}. This allocator is interesting, as it was at the origin of several `CVE` due to use-after-frees - see CVE-2016-9591, CVE-2016-9262 and CVE-2015-5221 for instance. These `CVE` requires to know the allocator (in these cases, it was retrieved manually by the authors).

`openssl` With `scat` on one of the testing binaries of `openssl`[4], we retrieve, with no option, the standard `libc` allocator. This is because every function from the `libc` which is used in `openssl` uses the standard allocator. However, if we ignore everything happening in the `libc`, we end up with the couple (`CRYPTO_malloc`,

[4] `hmactest in this case`

CRYPTO_free), which is a custom allocator embedded in openssl. The consistency rate of this allocator is equal to 1.13×10^{-25}.

jansson **[akh]** Jansson[6] uses its own allocator, (jsonp_malloc, jsonp_free), that scat successfully retrieve with a consistency rate of 0.00 (no error in 57 calls to ALLOC and 57 calls to FREE).

9.5.3 Membrush [CSB13]

The detection of allocators has not been subject to many researches. The most advanced existing work is [CSB13]. They also propose a heuristic-based approach in a dynamic context. Nevertheless, they use active instrumentation (and in particular, they replay the executions of a given function several times), which we exclude (as explained in Section 4.2.3). In addition, they base their approach on the tracking of the first allocation an allocator needs to perform (even custom ones): either through a system call mmap or brk, or through a standard allocator as malloc. It would have been interesting to compare our results with theirs. However, they use a proprietary benchmark (namely SPECINT 2006), and the source code of their implementation is not available. Finally, the authors do not provide measurements about overhead. Thus, it is hard to compare our two approaches.

[5] 47 errors were detected over 2111 calls to ALLOC and 2040 calls to FREE
[6] a C library for encoding, decoding and manipulating JSON data

	ALLOC/FREE	error rate	online (in s)	offline (in s)
b2sum	✓/✓	0	0.474	0.209/0.219
base32	✓/✓	0	0.561	0.121/0.119
base64	✓/✓	0	0.529	0.123/0.361
cat	✓/✓	0	0.48	0.177/0.199
chgrp	✓/✓	0.000397	0.521	0.143/0.132
chmod	✓/✓	0.000408	0.389	0.154/0.169
chown	✓/✓	5.15e-05	2.44	8.58/9.24
comm	✓/×	0.724	0.43	0.723/0.336
cp	✓/✓	0.000936	1.47	0.749/0.553
csplit	✓/✓	0	0.972	0.513/0.881
cut	✓/×	0.753	0.377	0.13/0.19
df	✓/×	0.27	0.677	0.677/0.658
dir	✓/×	0.0227	0.43	0.916/0.12
dircolors	✓/✓	0	0.407	0.244/0.291
du	✓/✓	0.000446	0.706	0.598/0.596
fmt	✓/✓	0	0.483	0.496/0.605
fold	✓/✓	0	0.569	0.129/0.134
ginstall	✓/✓	0	2.35	0.529/0.54
groups	✓/×	0.743	0.511	0.296/0.375
head	✓/✓	0	0.545	0.14/0.408
id	✓/×	0.743	0.533	0.282/0.373
join	✓/×	0.5	0.487	0.483/0.329
logname	✓/✓	0	0.657	0.314/0.439
ls	✓/×	0.0495	0.854	0.662/0.59
pinky	✓/✓	0	0.62	0.179/0.194
pr	✓/✓	0	0.652	0.504/0.289
rm	✓/✓	0.000198	20.5	0.268/0.268
shred	✓/✓	0	1.68	0.344/0.527
shuf	✓/×	0.949	0.431	0.109/0.179
sort	✓/✓	0	0.682	0.397/0.243
split	✓/✓	0	0.542	0.153/0.141
stat	✓/✓	0	0.791	0.203/0.198
stty	✓/✓	0	0.504	0.161/0.143
tail	✓/✓	0	0.501	0.143/0.2
tee	✓/✓	0	0.435	0.119/0.13
truncate	✓/✓	0	0.458	0.132/0.145
tsort	✓/×	0.428	0.395	0.325/0.638
uniq	✓/×	0.999	1.31	4.14/4.47
uptime	✓/✓	0	0.613	0.136/0.127
vdir	✓/×	0.0538	0.909	0.366/0.316
who	✓/✓	0.0259	0.684	0.178/0.142
whoami	✓/×	0.287	0.495	0.757/0.397

Table 9.6: General results of the allocator detection for programs using (`malloc`, `free`) on `coreutils`

	ALLOC/FREE	error rate	online (in s)	offline (in s)
8cc	n.c./n.c.	0	2.45	14/9.88
bash	×/×	0.13	4.12	0.497/0.224
git	n.c./n.c.	0	3.37	0.388/8.64e+04
grep	✓/×	0.885	0.857	0.711/0.648
mupdf-x11	✓/✓	0.0384	7.99	26.2/12.3
objdump	✓/×	0.978	1.03	0.676/0.453
openssl	✓/✓	0.00403	1.59	0.707/0.809
opusenc	×/×	0	1.6	1.17/0.966
readelf	n.c./n.c.	0	0.96	0.175/8.64e+04
strings	n.c./n.c.	0	0.424	0.233/8.64e+04
tar	✓/✓	0.000663	7.07	1.36/1.2
vim	✓/✓	0.402	32.9	1.11/0.764

Table 9.7: General results of the allocator detection for programs using (`malloc`, `free`) on the general benchmark

Conclusion

In this work, we proposed heuristic-based approaches to analyze binary programs from a single execution, at function-grained level. We aimed to retrieve structural information about functions, and behavioral information related to address data-flow, such as coupling and retrieving allocators. Our analyses do not rely on source code, and are applicable to stripped binaries.

We presented definitions, at binary level, of several notions that are usually source-level notions: functions, parameters and types. Although these concepts have already been defined and studied in previous works, they were expressed relatively to the source level. Instead, we proposed definitions that focus exclusively on the machine code, and yet that are consistent with source-level considerations. The interest of this is that we can express these concepts on binary code that is not obtained by compilation. From these definitions, we proposed heuristics to retrieve information from a single execution of binary programs. First, we have shown that we can retrieve arity of functions (*i.e.* the number of parameters they take) with simple heuristics, through one lightweight instrumentation. The main heuristic we used is to consider as a parameter a memory location that is read by a function before it has been written (since the function call). Second, we proposed a way to retrieve *undertypes* of each parameter, *i.e.*, to differentiate parameters holding addresses from parameters holding numeric values. The central heuristic of this part is to consider as an address every concrete value used as a memory operand at some point of the execution (not necessarily in the considered function). Third, we introduced a notion of coupling which is useful to retrieve allocators of resources. Two functions are coupled with a coupling rate ρ if parameter values of one function come from the output of the other in a proportion of at least ρ. We also proposed a statistic method to retrieve coupling from a single execution.

Finally, we presented heuristics to retrieve efficiently the main memory allocator in a binary program. The major heuristic of this step consists in identifying as an allocator a function that outputs a lot of memory addresses, and the corresponding deallocator as the last accessor of an allocated memory block. In addition, we proposed a metric, named *consistency rate*, to confirm or deny the result of the allocator analysis.

Our experimental results show that the approach is indeed accurate and scalable. Relatively to accuracy, we present a rate of 93% for arity detection, and for undertype detection. For allocators, we have shown that our approach is able to retrieve the standard `libc` allocator on many programs, as well as custom embedded allocators in several examples (`openssl`, `jasper`, etc.). We have also shown that our consistency rate is a very good indicator to validate the detection when it is indeed correct, and to detect errors when it is not. From a scalability point of view, we presented the overhead of our implementation for each step. We have shown that the execution time remains limited with common programs such as `git` or `mupdf`. However, using `Pin` to perform dynamic instrumentation leads to an important overhead in percentage, especially on small programs executing fast (*e.g.*, `coreutils` programs).

This work could be extended in several directions. First, we propose experiments to test our approach in various contexts. It would be interesting to perform more experiments, to confirm several points. Regarding the low coverage[7] (because we work on a single execution), we could use `scat` with a fuzzer, such as `AFL`, to improve the number of functions we detect (and because we perform a lightweight instrumentation, this seems practical). In addition, the approach should be tested with other compilers than `gcc` and on other architectures than `x86-64` to ensure the universality of our method in practice.

Second, we propose some perspectives to extend the approach, and we discuss the possible applications of our results.

From a global point of view, our approach focuses on imperative programming, with few considerations about object-oriented programming and functional programming. As a perspective, it would be interesting to see how it can be adapted to such paradigms.

Our definition of arity does not include variadic functions (*i.e.* functions with a

[7] *i.e.*, we only detect a few number of functions

variable number of parameters). It could be extended, to detect parameters that are not always given during a call. However, such an approach would be delicate to develop with respect to our criteria.

The notion of types we target is a significant restriction of the types we express at source level. In particular, we do not include the notion of size in our subtypes. Detecting size of data is a challenging issue: at assembly level, operations are often performed on the whole registers (*e.g.* on 64 bits) although the relevant data is only one byte long. Such analyses would require a refinement of data flow that is more precise that the one we proposed at the grain of functions. More generally, we could extend our approach to focus on a grain of data structures.

We introduced a notion of coupling, that we use to detect allocators. However, this notion is general, and could be interesting in other contexts. For instance, finding transitive couples (f coupled with g and g coupled with h) may be useful to have an overview of the behavior of a program.

Regarding the detection of allocators, the perspectives are manifold. First, our approach leads to the detection of the *most frequently used* allocator, but there might be other allocators embedded in a given program. A future work would be to detect *every* allocator in a binary, and to retrieve a hierarchy, how they interact and what part of the program uses which one. Second, we target `ALLOC` and `FREE`, but an allocator often presents other methods, and in particular `REALLOC`. The detection of `REALLOC` would be very interesting to understand how the program reuses memory. Third, the detection of custom allocators allow to perform memory checks for safety and security. In particular, detecting use-after-free bugs is a trending research subject, but many analyses rely on the knowledge of the allocator. The combination of our approach to detect allocators with analyses of this type can lead to detect new bugs and vulnerabilities in programs using a custom allocator.

To conclude, this work shows that it is possible to perform accurate analyses on common programs in one instrumented execution. In particular, a lightweight instrumentation can lead to retrieve prototypes of functions, coupled functions and memory allocators with a good accuracy and a minimum of false positives. It also shows that heuristic-based approaches lead to an implementation that is scalable and usable in practice.

> *Whereof one cannot speak, thereof one must be silent.* – Ludwig Wittgenstein

Bibliography

[AF11] Christoph Alme and Stefan Finke. Predictive heap overflow protection, December 9 2011. US Patent App. 13/315,928.

[akh] akheron. Jansson. `https://github.com/akheron/jansson`.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. pages 259–269, 2014.

[ASB17] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *IEEE European Symposium on Security and Privacy*, 2017.

[AT15] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. Detection and identification of android malware based on information flow monitoring. In *IEEE 2nd International Conference on Cyber Security and Cloud Computing, CSCloud 2015, New York, NY, USA, November 3-5, 2015*, pages 200–203. IEEE, 2015.

[BBW+14] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: learning to recognize functions in binary code. pages 845–860, 2014.

[BD06] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. *Black Hat Europe*, 6:25–47, 2006.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 1–12. ACM, 2000.

[Ber06] Emery D Berger. Heapshield: Library-based heap overflow protection for free. *UMass CS TR*, pages 06–28, 2006.

[bin] The gnu binutils - a collection of binary tools. `https://www.gnu.org/software/binutils/`.

[BK12] Edd Barrett and Andy King. Range analysis of binaries with minimal effort. In Mariëlle Stoelinga and Ralf Pinger, editors, *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, volume 7437 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2012.

[BMKK06] Ulrich Bayer, Andreas Moser, Christopher Krügel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[BR04] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004.

[BR07] Gogul Balakrishnan and Thomas W. Reps. DIVINE: discovering variables IN executables. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2007.

[BR10] Gogul Balakrishnan and Thomas W. Reps. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010.

[BRTV16] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. Statistical deobfuscation of android applications. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 343–355. ACM, 2016.

[BST00] Arash Baratloo, Navjot Singh, and Timothy K. Tsai. Transparent run-time defense against stack-smashing attacks. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 251–262. USENIX, 2000.

[CAM+08] Xu Chen, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, pages 177–186. IEEE Computer Society, 2008.

[CBP+11] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[CCT+15] Chien-Ming Chen, Shuai-Min Chen, Wei-Chih Ting, Chi-Yi Kao, and Hung-Min Sun. An enhancement of return address stack for security. *Computer Standards & Interfaces*, 38:17–24, 2015.

[CG95] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[CGMN12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 133–143. ACM, 2012.

[CJMS10] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, 2010.

[CJS+05] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA*, pages 32–46. IEEE Computer Society, 2005.

[Cor] Microsoft Corporation. Windbg. `http://www.windbg.org/`.

[Cow98] Crispan Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Aviel D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998.

[CSB13] Xi Chen, Asia Slowinska, and Herbert Bos. Who allocated my memory? detecting custom memory allocators in C binaries. In Ralf Lämmel, Rocco Oliveto, and Romain Robbes, editors, *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 22–31. IEEE Computer Society, 2013.

[CTL98] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 184–196. ACM, 1998.

[CW08] Brian Chess and Jacob West. Dynamic taint propagation: Finding vulnerabilities without attacking. *Inf. Sec. Techn. Report*, 13(1):33–39, 2008.

[CZW14] Bing Chen, Qingkai Zeng, and Weiguang Wang. Crashmaker: an improved binary concolic testing tool for vulnerability detection. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong, editors, *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1257–1263. ACM, 2014.

[DBL11] *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[Des] Fabrice Desclaux. Miasm: Framework de reverse engineering.

[DG11] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, pages 77–101, 2011.

[dGFM17] Franck de Goër, Christopher Ferreira, and Laurent Mounier. scat: Learning from a single execution of a binary. In Pinzger et al. [PBM17], pages 492–496.

[EAK+13] Khaled Elwazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 51–60. ACM, 2013.

[EGV16] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In Brecht Wyseur and Bjorn De Sutter, editors, *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Vienna, Austria, October 24-28, 2016*, pages 27–38. ACM, 2016.

[FMP14] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *J. Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[Foua] Free Software Foundation. Gnu gcov. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[Foub] Free Software Foundation. Gnu gprof. `https://sourceware.org/binutils/docs/gprof/`.

[Fouc] Free Software Foundation. Objdump. `https://sourceware.org/binutils/docs-2.21/binutils/objdump.html`.

[FZPZ08] Wen Fu, Rongcai Zhao, Jianmin Pang, and Jingbo Zhang. Recovering variable-argument functions from binary executables. In Roger Y. Lee,

editor, *7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008, 14-16 May 2008, Portland, Oregon, USA*, pages 545–550. IEEE Computer Society, 2008.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.

[git] Git - fast version control. `https://git-scm.com/`.

[GMS+15] Marco Gaudesi, Andrea Marcelli, Ernesto Sánchez, Giovanni Squillero, and Alberto Paolo Tonda. Malware obfuscation through evolutionary packers. In Sara Silva and Anna Isabel Esparcia-Alcázar, editors, *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 757–758. ACM, 2015.

[gre] Grep. `http://www.gnu.org/software/grep/`.

[GSS90] John Gilmore, Cygnus Solutions, and Stan Shebs. gdb internals, 1990.

[Gut] Peter Gutmann. Fuzzing code with afl.

[Hee09] Sean Heelan. Finding use-after-free bugs with static analysis. https://sean.heelan.io/2009/11/30/finding-bugs-with-static-analysis/, 2009.

[HGOR13] Karim Hossen, Roland Groz, Catherine Oriat, and Jean-Luc Richier. Automatic generation of test drivers for model inference of web applications. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 441–444. IEEE Computer Society, 2013.

[HGR11] Karim Hossen, Roland Groz, and Jean-Luc Richier. Security vulnerabilities detection using model inference for applications and security

protocols. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 534–536. IEEE Computer Society, 2011.

[How02] Michael Howard. Some bad news and some good news. `http://msdn.microsoft.com/library/default.asp?url=/library/enus/dncode/html/secure10102002.asp`, October 2002.

[HR] SA Hex Rays. Ida pro disassembler and debugger. `https://www.hex-rays.com/products/ida/`.

[HSJC12] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012.

[JCG+14] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In Suresh Jagannathan and Peter Sewell, editors, *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2014, PPREW 2014, January 25, 2014, San Diego, CA*, pages 1:1–1:11. ACM, 2014.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 226–235. ACM, 1999.

[KSZ+14] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on*

Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, pages 1128–1139. ACM, 2014.

[KW09] Wen-Fu Kao and Shyhtsun Felix Wu. Lightweight hardware return address and stack frame tracking to prevent function return address attack. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009, Vancouver, BC, Canada, August 29-31, 2009*, pages 859–866. IEEE Computer Society, 2009.

[LAB11] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011* [DBL11].

[LCM+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. pages 190–200, 2005.

[Lin05] Christian Lindig. Random testing of C calling conventions. In Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, Monterey, California, USA, September 19-21, 2005*, pages 3–12. ACM, 2005.

[Luk] Dejan Lukan. Linear sweep vs recursive disassembling algorithm. `http://resources.infosecinstitute.com/linear-sweep-vs-recursive-disassembling-algorithm/#gref`.

[MHJM13] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. amd architecture processor supplement. *Also available as http://x86-64.org/documentation/abi.pdf*, 2013.

[MM16] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 24–35. ACM, 2016.

[Moo56] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[mor] Morris worm. `http://www.ee.ryerson.ca/~elf/hack/iworm.html`.

[mup] Mupdf. `http://mupdf.com/`.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003.

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. pages 89–100, 2007.

[NYH+15] Phu Hong Nguyen, Koen Yskout, Thomas Heyman, Jacques Klein, Riccardo Scandariato, and Yves Le Traon. Sospa: A system of security design patterns for systematically engineering secure systems. In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 246–255. IEEE Computer Society, 2015.

[OMR04] MFXJ Oberhumer, László Molnár, and John F Reiser. Upx: the ultimate packer for executables, 2004.

[ope] Openssl - cryptography and ssl/tls toolkit. `https://www.openssl.org/`.

[opu] Opus interactive audio codec. `http://www.opus-codec.org/`.

[osa] GNU operating system. Bash - bourne again shell. `https://www.gnu.org/software/bash/`.

[osb] GNU operating system. Coreutils - gnu core utilities. `https://www.gnu.org/software/coreutils/coreutils.html`.

[osc] GNU operating system. Gnu tar. `https://www.gnu.org/software/tar/tar.html`.

[OVB+06] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Computers*, 55(10):1271–1285, 2006.

[PBM17] Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors. *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. IEEE Computer Society, 2017.

[Pol70] Polybe. *Histoire*. Number (20) - (23) in 1. Gallimard, 1970.

[PW97] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In Steve Vinoski, editor, *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS), June 16-20, 1997, Portland, Oregon, USA*, pages 185–198. USENIX, 1997.

[QLZ05] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, pages 291–302. IEEE Computer Society, 2005.

[RB08] Thomas W. Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In Laurie J. Hendren, editor, *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4959 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2008.

[RILR14] Amira Radhouani, Akram Idani, Yves Ledru, and Narjes Ben Rajeb. Extraction of insider attack scenarios from a formal information system

modeling. In Véronique Cortier and Riadh Robbana, editors, *Proceedings of the Formal Methods for Security Workshop co-located with the PetriNets-2014 Conference , Tunis, Tunisia, June 23rd, 2014.*, volume 1158 of *CEUR Workshop Proceedings*, pages 5–19. CEUR-WS.org, 2014.

[RKMV03] William K. Robertson, Christopher Krügel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In Æleen Frisch, editor, *Proceedings of the 17th Conference on Systems Administration (LISA 2003), San Diego, California, USA, October 26-31, 2003*, pages 51–60. USENIX, 2003.

[RM12] Sanjay Rawat and Laurent Mounier. Finding buffer overflow inducing loops in binary executables. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, pages 177–186. IEEE, 2012.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010.

[SDA02] Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. Disassembly of executable code revisited. In Arie van Deursen and Elizabeth Burd, editors, *9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*, pages 45–54. IEEE Computer Society, 2002.

[Sha08] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Components to support Integration Testing*. PhD thesis, PhD thesis, Grenoble Institute of Technology, 2008.

[sla] Slammer worm. `https://www.wired.com/2003/07/slammer/`.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall,

editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[SO06] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 123–134. IEEE Computer Society, 2006.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011* [DBL11].

[Uey12] Rui Ueyama. 8cc: a small c compiler. `https://github.com/rui314/8cc`, 2012.

[VCKL05] Michael Venable, Mohamed R. Chouchane, Md. Enamul Karim, and Arun Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In Klaus Julisch and Christopher Krügel, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005, Proceedings*, volume 3548 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[vim] Vim - the ubiquitous text editor. `http://www.vim.org/`.

[WPV17] Xiaoran Wang, Lori L. Pollock, and K. Vijay-Shanker. Automatically generating natural language descriptions for object-related statement sequences. In Pinzger et al. [PBM17], pages 205–216.

[XMW16] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized dynamic opaque predicates: A new control flow obfuscation method. In Matt Bishop and Anderson C. A. Nascimento, editors, *Information Security - 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016, Proceedings*, volume 9866 of *Lecture Notes in Computer Science*, pages 323–342. Springer, 2016.

[You15] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[Yus] Oleh Yuschuk. Ollydbg. `http://www.ollydbg.de/`.