Finding the needle in the heap : combining binary analysis techniques to trigger use-after-free

Josselin Feist

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministérial : 25 mai 2016

Présentée par

Josselin Feist

Thèse dirigée par **Marie-Laure Potet and Laurent Mounier**

préparée au sein **Verimag**
et de **École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Finding the Needle in the Heap: Combining Binary Analysis Techniques to Trigger Use-After-Free

Thèse soutenue publiquement le **29 mars 2017**,
devant le jury composé de :

PHILIPPE ELBAZ-VINCENT
Professeur, Université Grenoble Alpes, Président
YVES LE TRAON
Professeur, Université du Luxembourg, Rapporteur
VINCENT NICOMETTE
Professeur, INSA Toulouse, Rapporteur
PASCAL CUOQ
Directeur Scientifique, TrustInSoft - Paris, Examinateur
AURELIEN FRANCILLON
Maitre de Conférences, Eurecom - Sophia-Antipolis, Examinateur
SARAH ZENNOU
Chef de Projets R&D, Airbus Group Innovations - Suresnes, Examinatrice
MARIE-LAURE POTET
Professeur, Grenoble INP, Directrice de thèse
LAURENT MOUNIER
Maître de Confrences, Université Grenoble Alpes, Co-Directeur de thèse

Abstract

Security is becoming a major concern in software development, both for software editors, end-users, and government agencies. A typical problem is vulnerability detection, which consists in finding in a code, bugs able to let an attacker gain some unforeseen privileges like reading or writing sensible data, or even hijacking the program execution. The problem of finding of vulnerabilities is, therefore, a major concern, but detecting them in software designed with no security rules is an arduous task. The use of automated program analysis techniques helps security researchers to reduce the effort of detection.

This thesis proposes a practical approach detecting a specific type of vulnerability, called use-after-free, which appears when a heap memory block is used after being freed. Use-after-free vulnerabilities were the origin of numerous recent exploits, most often in web browsers. Such flaws are complex to find and need dedicated methods and models to be well taken into account. Current systems detecting them are generally based on one specific method and are either too narrow to find the complex pattern behind their triggering or are too costly to scale and be applied for a security purpose.

In this thesis, we propose to combine two well-known program analysis techniques, static analysis and dynamic symbolic execution, in order to benefit from the strengths of each one without their defects. More precisely, we design a coarse-grain and unsound static analyzer, called GUEB, built to be efficient when applied to binary code. We propose memory models well-adapted to follow the heap operations (allocation, free and use) involved in use-after-free. This leads to a program slice containing potential candidates to be further analyzed. Our static analysis demonstrates its strengths by finding previously undiscovered vulnerabilities in several real-world applications. In the second step of the approach, dedicated guided dynamic symbolic execution, developed within the Binsec platform, is used to retrieve concrete program inputs aiming to trigger these use-after-free. While dynamic symbolic execution tends to get lost in the many possible execution paths of a program, this combination allows to guide the exploration directly toward triggering use-after-free. The effectiveness of this association is demonstrated by its application to a real-world program. Our approach is thereby both scalable and precise enough to be applied on real binaries. All the tool-chain is implemented into open-source software operating on x86 binary code.

Remerciements

Cette thèse, qui est le fruit d'un travail sur plusieurs années, n'aurait pas pu aboutir sans l'aide et le soutien de nombreuses personnes.

En premier lieu, je tiens à remercier mes encadrants, Marie-Laure Potet et Laurent Mounier, pour m'avoir apporté leurs expériences et leurs conseils tout au long de ces années. De plus j'exprime ma gratidue à l'ensemble de mon jury, Philippe Elbaz-Vincent, Pascal Cuoq, Aurélien Francillon, Yves Le Traon, Vincent Nicomette et Sarah Zennou, qui par la pertincence de leurs retours et questions, m'ont permis d'améliorer ce manuscrit.

Mes remerciements vont naturellement aussi à tous mes collègues et amis de Verimag. En particulier à Gustavo Grieco, qui a su dès le début m'aider et partager ses connaissances, Ozgun Pinarer, sans qui la thèse aurait été une épreuve bien plus difficile, Mariuca et Mihail Asavoae, avec qui j'ai savouré de nombreux moments de détente (et des muffins), Louis Dureuil, avec qui j'ai exploré l'aventure qu'est le doctorat, Ta Thanh Dinh, pour l'enrichissement qu'ont été nos discussions et Maxime Puys, qui a su me supporter durant mes derniers mois en tant que doctorant. J'ai eu la chance de collaborer à de nombreuses reprises avec le CEA LIST et mes travaux n'auraient pas eu la même qualité sans l'aide de Sébastien Bardin et Robin David, que je remercie.

J'ai eu l'opportunité de participer activement à Securimag durant mes années de thèse. Je remercie donc toutes les personnes qui font vivre l'association. La liste de remerciements serait trop longue, mais je tiens à remercier Karim Hossen pour son aide à la fois à Securimag et dans mes travaux, ainsi que Franck De Goër pour avoir accepté de gérer l'association lorsque je ne pouvais plus le faire. Je suis aussi reconnaissant envers Florent Autreau, qui a su me soutenir dans de nombreux projets. Sans son aide, ces dernières années auraient été bien plus difficiles.

Ce manuscrit n'aurait pas eu la même qualité sans l'aide d'Irina Nicolae, que je remercie aussi pour son soutien durant ces longues périodes. Enfin, j'ai une pensée pour tous mes proches et ma famille, qui m'ont toujours supporté et épaulé. En particulier, je remercie mes parents, qui par leurs efforts sans fin, ont permis à mes frères, ma soeur et moi-même, d'avoir la liberté dans nos choix et nos décisions de vie.

Contents

Chapter 1

Introduction

1.1 Program Vulnerabilities

A computer program is designed to perform a set of particular tasks, and its behavior depends on the inputs which are given to it. In a perfect world, programs would always react in a way that was anticipated by its developers and expected by the users. Unfortunately, in the real world programs contain bugs, and therefore, sometimes a program does not behave as expected. Since the very beginning of the computer science [Sha87], developers have inadvertently designed programs containing errors. While some industries, in particular those working with critical systems, invested an important amount of money and effort to design reliable software and to verify and secure their components, most developers do not follow strict rules when programming. As a consequence, the number of bugs in software in general is enormous. For example, at the time of writing this thesis the number of open bugs in Ubuntu is more than 120,000 [Ubu]. Fixing a bug is time consuming, and with such a large number of bugs, fixing all of them is not realistic. Bugs can be categorized according to their impacts, we can distinguish:

- Benign bugs (with almost no impact on the program behavior);
- Bugs alternating the program functionalities, or performances;
- Bugs decreasing the security of a system called vulnerabilities.

This thesis focuses on vulnerabilities. OWASP defines a vulnerability as [OWA]:

> A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application.

There are several categories of vulnerabilities. For example, an error in the design of a cryptographic algorithm could yield to make the encryption useless. In a different approach, *injection attacks*, such as *SQL injections*, are present because of a lack of verification of the user inputs that are interpreted at some point as a command or a query. Such injection attacks can result in the reading, the modification of system data or even the executing of a command in the system directly. A particular class of vulnerabilities focuses on errors occurring in the memory associated to a program; these are called *memory corruptions*. The vulnerability studied in this thesis belongs to this category. Exploiting a memory corruption can result in serious consequences, of which the most typical are:

- The system becomes unavailable (e.g.: *denial-of-service* attacks);
- Critical information is leaked (such as cryptographic keys);
- In the worst case scenario, the full system can be compromised (e.g.: if the exploitation leads to execute unauthorized code).

Memory corruptions are also commonly used by malware to infect hosts and allow them to spread to all unfixed systems. Some vulnerabilities became well known these past years because of their impact; for example, Dirty Cow [Gooa], Stagefright [Fin] or Heartbleed [McM] are recent

vulnerabilities, impacting a large number of systems, and they have even attracted the attention of the general information media.

Vulnerabilities have a cost for the developer. Finding how to fix them takes time, and moreover, deploying the appropriate fix can be costly. Moreover, as they impact the users of the program, they can also impact the public image of the developer or of the company and reduce the confidence of the users in its products. Yet, the real cost of unfixed vulnerabilities is difficult to evaluate. On the other hand, nowadays there is a market for selling and buying vulnerabilities, and the price paid for them is constantly rising. For example, more and more companies provide compensation for reporting vulnerabilities through *bug bounty* programs. Some companies (such as Zerodium[1]) have even based their business model on purchasing and reselling vulnerabilities, and are sometimes willing to pay millions for some of them. Vulnerability market no longer concerns only software editors, but also government agencies and armed forces.

Due to the dangers encompassed by the use of programs containing vulnerabilities, the capacity to find them is an important topic.

1.2 Automated Vulnerabilities Discovery: Background

Manually finding vulnerabilities is complex and time-consuming. Nowadays, security researchers use program analysis techniques to help them with the discovery of vulnerabilities. These techniques are classically split into two categories:

1. Dynamic analysis, such as fuzzers [SGA07]. The idea is to generate inputs used to stress the program, thus triggering vulnerabilities;

2. Static analysis, which consists of analyzing the program without actually executing it. Static analysis aims to model possible behaviors that could arise at runtime.

By creating inputs, fuzzers explore concrete executions of the program. As the number of possible different executions could be significant, and because fuzzers are applied in a finite time, such analyses lead to explore only a part of possible behaviors of the tested program. Fuzzers thereby belong to the family of analyses performing *under-approximation* of possible programs behaviors, which do not find all bugs. Yet, in the security industry, fuzzers are widely used and show a practical efficiency to detect vulnerabilities. For example, open source fuzzers have a large list of found vulnerabilities, of which we mention AFL [Zal], radamsa [Hel], quickfuzz [GCB].

Static analysis is widely used as a verification technique; in particular, one key feature of the static analysis is that it to prove the absence of bugs. However, static analysis has one major issue: the number of false alarms raised. Most of the static analyzers are indeed based on an *over-approximation* of possible behaviors of the program. This over-approximation leads to consider possible paths that are not feasible in reality. Thereby, most tools based on static analysis claim to find vulnerabilities where there are none. Moreover, due to its nature, static analysis is limited by undecidable problems [Lan02]; hypotheses and restrictions are then needed. The adoption of static analysis in the security community is, therefore, more difficult.

During the last decade, we have witnessed the emergence of new techniques trying to reach a compromise between static and dynamic analysis. Dynamic symbolic execution (DSE) is one of them. DSE is a program path exploration method which combines a dynamic execution of the program with a symbolic reasoning. This technique generates an input for each explored path and, similarly to fuzzers, only explores a part of the possible behaviors of the programs.

Each program analysis technique has its strengths and its weaknesses; more and more methods try to combine different techniques to benefit from their strengths without suffering from their weaknesses.

Necessity of binary analysis In practice, the source code of a program is not always available. Moreover, while the source code can be available, it is not necessary the case for its libraries. The capacity to provide program analysis techniques working directly on the binary code becomes essential in that case. Dynamic analysis by its nature generally works on the binary version of the program. Static analysis, on the other hand, is in most cases designed to work on source code. While working at the binary level brings several difficulties for the analysis (as some information is lost), it also has its advantages:

[1] https://www.zerodium.com

- Programs can be written using *undefined behaviors* of the language. Two different compilers can then produce two different binaries with different behaviors. By analyzing the binary code, we take into account the actual interpretation of these undefined behaviors;

- The combination of different techniques (such as static and dynamic analysis) is easier when applied on a binary; we do not rely on any assumptions or representations of the source code, the code analyzed is the same all along the way.

1.3 Thesis Objectives

Some classes of memory corruptions, such as buffer overflow or integer overflow, have been well studied. On the contrary, recent classes of vulnerability, such as Use-After-Free or type confusion, did not receive as much attention. This thesis focuses on the former: Use-After-Free.

More precisely, it aims at triggering Use-After-Free using automated program analysis techniques on binary codes. Triggering and not only detecting it is a requirement to make automated program analysis attractive for security researchers, because by actually triggering a vulnerability we demonstrate its reality. To achieve such a goal, we base our approach on the combination of two formal methods: static analysis and dynamic symbolic execution.

Using formal methods in security This thesis aims to fill the gap between theory and practice. Therefore, a particular effort has been put into adapting our approach to real-world examples. We choose two classic formal methods, static analysis and dynamic symbolic execution, and study how they can be used for security purposes applied to Use-After-Free. We design several heuristics based on practical observations, making our analysis applicable to realistic contexts. Thereby, our contribution falls in the category of the experimental science; we lose the soundness of the standard formal methods, but the applicability of our method is one of our primary motivations.

Goals in static analysis As mentioned before, static analysis is a powerful method, yet its adoption for security purpose is still limited. Moreover, Use-After-Free is a vulnerability occurring in the heap. Modeling such a structure is a complex task for static analysis; handling it generally comes with a cost on the approximation. Our work on static analysis can be split into two axes. The first one focuses on how to design a static analyzer working at the binary code level that keeps an acceptable trade-off between precision and scalability. The second tackles the heap model and the actual Use-After-Free detection. More precisely, our method is suitable to detect vulnerabilities with an acceptable number of false positives, which is demonstrated by the previously unknown vulnerabilities found on six different real-world programs using our static analyzer.

Goals in dynamic symbolic execution Dynamic symbolic execution is an active field of research. Yet, using such a method in real-world applications is still challenging, and tools based on this technique generally need tuning to become applicable. In this thesis, we enhance dynamic symbolic execution by using information coming from static analysis. This combination allows to quickly trigger the targeted vulnerability. We also provide several refining heuristics for dynamic symbolic execution that we develop, making our tools working on real-world programs. To demonstrate the applicability of our approach, we use it to create an input triggering a Use-After-Free corresponding to a previously unknown vulnerability found by our static analyzer.

Open source tools and reproducible experiments While automated vulnerability-finding tools have been an active area of research, the security community still lacks available solutions. Most tools are never published, yet to be effective, program analysis tools need a consequent effort of engineering. Moreover, to be effective, the experimental science needs a way to reproduce the published experiments; yet, as tools are never published, experiments are generally not reproducible. By sharing tools as open-source project, and providing all files for our experiments, we hope to actively participate in the opening of the vulnerability detection activity and to promote a better access to such techniques.

1.4 Thesis Outline

This thesis is split into four parts:

- First, Part I explains Use-After-Free and gives an overview of our approach triggering such a vulnerability on binary code. It introduces the topic studied in this dissertation and provides a detailed state of the art of techniques used to detect or prevent Use-After-Free. Then an overview of our global approach combining static analysis with dynamic symbolic execution is illustrated through the use of a motivating example.
- Part II focuses on the static analysis built to detect Use-After-Free. It details models used by the static analysis and formalizes our contributions on the detection of this vulnerability. Information on the implementation of the static analysis is provided, as well as benchmarks on real-world examples, showing the efficiency of our approach.
- Then, Part III details the guided dynamic symbolic execution developed to be used in collaboration with the static analysis. This combination is the core of our approach. In addition, this part gives several contributions making dynamic symbolic execution more efficient when applying to the real-world context.
- Finally, Part IV concludes this document with a discussion on our contributions and possible extensions of this work.

Working hypotheses We make the following assumptions in the remainder of this document:

- Binaries analyzed are built from a C program, using a classic compiler. Binaries do not contain obfuscations or self-modifying techniques;
- The architecture of the binaries analyzed is the *x86* of Intel, yet our contributions can be adapted to other architectures;
- By working directly on binary codes, our techniques also work on programs written in C++, but are less efficient in such programs.

Part I

Triggering Use-After-Free: Concepts and Contributions

Chapter 2

Heap Vulnerabilities

This chapter aims to introduce some notions that are needed to fully understand the topic studied in this thesis. Among memory corruptions, a particular class of vulnerability concerns errors occurring in the heap. These vulnerabilities differ from others in that they are related to the dynamic memory management. While common memory corruption, like buffer overflow or use of uninitialized memory may occur in both heap memory and non-heap memory, the dynamic memory management brings new types of vulnerabilities and new types of exploitations.

Outline The remainder of this chapter is organized as follows:

- Section 2.1 recalls some notions on the memory of a program and details problems related to the analysis of heap vulnerabilities;
- Section 2.2 details the vulnerability studied in this thesis: Use-After-Free;
- Finally, Section 2.3 describes the state of the art of techniques and tools detecting or protecting against this vulnerability.

2.1 Heap Particularities

This section provides notions and concepts needed to understand a program analysis dealing with the heap.

2.1.1 Memory Layout

The memory layout of a program is split into several sections. We generally distinguish five sections (represented in Figure 2.1).

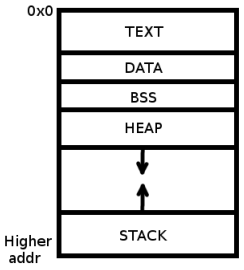


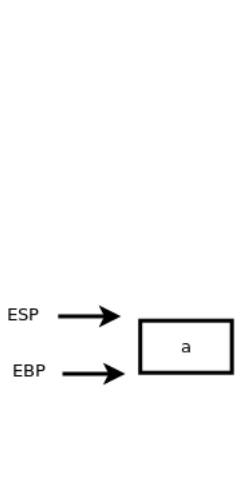
Figure 2.1: Memory sections representation.

The *TEXT* section contains the executed instructions of the program. In the *DATA* section, `constant` variables used by the program are located. *BSS* contains both `static` and `global` variables. Dynamic allocations are located inside the *HEAP* and are described below. Finally, the *STACK* is the memory section where local variables of a function are stored.

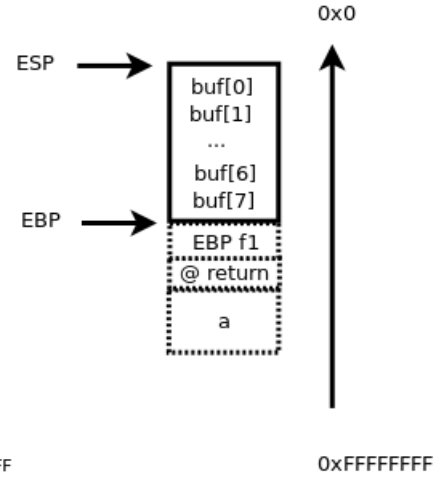
Local variables Whenever a function is called, the memory needed for its local variables is allocated in the stack. The stack grows toward lower addresses, meaning that the last allocated variable has the lowest address. The part of the stack that belongs to a function is called its *stack frame*. Since the number of local variables of a function is finite, the size of the *stack frame* is computed during compilation[1]. In the *x86* architecture, the stack frame of the current function is delimited by two specific registers: *ebp* and *esp*, respectively for the base pointer and the stack pointer. The former points to the beginning of the stack frame, while the latter refers to its end (so *ebp* has a higher value than *esp*). Some other specific values are kept in stack frames, like function parameters, the address of the instruction to be executed after the return of the function, or the value of *ebp* corresponding to the *stack frame* of the caller. Figure 2.2 shows an example of a stack frame layout. Here a function `f1` contains one local variable `a`; its stack frame is represented in Figure 2.2b. During the call to `f2`, the space for the local variable `buf[8]` is allocated in the stack, as well as the return address of `f2` and the previous value of *ebp*, as represented in Figure 2.2c. Notice that the address of the instruction to be executed after the return, which is saved between two *stack frames*, is the main target to exploit a program in a context of a buffer overflow, since the modification of this value allows to control the execution flow directly.

```
1  void f2(){
2   char buf[8];
3  }
4
5  void f1(){
6   int a;
7   f2();
8  }
```

(a) Source code.



(b) Before f2()



(c) During f2()

Figure 2.2: Stack frame example.

Dynamic variables Dynamic variables (also called heap variables) are allocated in the *HEAP* section during the execution. Contrary to local variables, their sizes can be computed at runtime, and this memory section grows toward higher addresses. Functions such as *malloc* or *calloc* in the *glibc* library allocate such variables.

2.1.2 Allocator Strategies

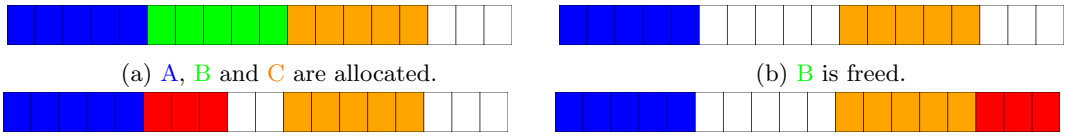
The memory of a computer available for a program is finite; thus the stack and the heap are bounded. Parts of the memory which are no longer used need to be released in order to be available for other allocations. For the stack, this is done implicitly when the function returns: the *stack frame* of the function is released. For the heap, it can either be done automatically, using garbage collectors or let to the responsibility of the developer. From a security point of view, garbage collectors provide a safer dynamic memory management scheme, yet they bring an overhead which makes them not suitable for all purposes. A comparison of performance between garbage collector and manual management of the heap can be found in [HB06]. Functions managing the release of dynamic variables are provided by standard libraries, such as *free* in the *glibc* library. The C11 standard [INC12a] refers:

[1]Notice that some function, as `alloca`, can dynamically allocate new variables to the *stack frame*, which changes its size.

> The order and contiguity of storage allocated by successive calls to the *aligned_alloc*, *calloc*, *malloc*, and *realloc* functions is unspecified.

Thus, there is no specification on the way dynamic memory should be handled. Several allocation strategies exist, which differ by the trade-off between their speed and the memory space needed. Some are also more adapted when encountering specific properties, such as multi-thread programming, or real-time. To understand how they can differ, we consider a common problem when dealing with the heap: fragmentation.

Fragmentation Figure 2.3a represents a heap memory constituted of 18 possible blocks, where three successive allocations of five blocks are performed: A, B and C.



(a) A, B and C are allocated.

(b) B is freed.

(c) Naive allocator.

(d) Best fit allocator.

Figure 2.3: Heap fragmentation example.

In the next step, the chunk B is freed, as represented in Figure 2.3b. Assume that now we want to allocate a block of size 3. If we consider a *naive* allocator, which returns the first location with enough space available, we obtain a heap with the state represented in Figure 2.3c. Such a decision prevents a future allocation of a size greater than 3. If we consider now a smarter, but slower, strategy which tries to find the location that fits best the size requested, the heap is in the state represented in Figure 2.3d. While in this case an allocation with a size up to 5 can be done, the time spent to find the right location is longer. When dealing with real applications, this trade-off is crucial, and thus the right strategy greatly depends on the application. Studies on the design and the performance of several allocation strategies can be found in [Wil+95; Fer+11].

Standard allocator To illustrate the complexity of modeling allocation strategies, we give here an overview of *ptmallocv2*, which is used by the *glibc*. *ptmallocv2* is based on *dlmalloc* [Lea00]. One of the major improvements compared to *dlmalloc* is the support for multi-thread applications. The following explanation details both *ptmallocv2* and *dlmalloc*, except for the thread mechanism. Each chunk allocated by these strategies contains metadata, located before the usable area of the chunk. Thereby, when `malloc` returns the address X, the metadata is located at $X - size(metadata)$. This metadata contains, among other, the size of the chunk. When a chunk is freed, it is placed into a free list. Free lists are used to keep track of freed chunks and are handled through linked lists. This free list is stored directly into the freed chunks, by writing over user data. Notice that this technique can be useful for exploitation: writing on a freed pointer will re-write these addresses, corrupting the free list. Free chunks are grouped by their size and are kept in *bins*. There are several bins: fast bin (chunks of size < 80 bytes), unsorted bin (chunks recently freed), small bin (chunks of size > 80 and < 512 bytes), and large bin (chunks of size > 512 bytes). Each bin has it own data-structure (fast bin uses single linked list, while others use circular double linked list), and specific behaviors (in term of splitting, coalescing, etc.). To handle multi-threading applications (in *ptmallocv2*), each thread holds its heap segment and its free lists; this mechanism is called per thread arena. More details on *ptmallocv2* can be found in [Fer07; FFM12; spl15].

As the previous example shows, allocators are complex systems; they combine different structures and behaviors. There exist several standard allocation strategies, including:

- *phkmalloc* used by OpenBSD [Kam98];
- *tcmalloc* developed by Google and used in Chrome[San];
- *jemalloc* used by FreeBSD[Eva06] (and by Facebook [Eva11]);
- *Low Fragmentation Heap* used in Microsoft Windows[Val10; RSI12];
- *Hoard*, designed to work on several operating systems[Ber+00].

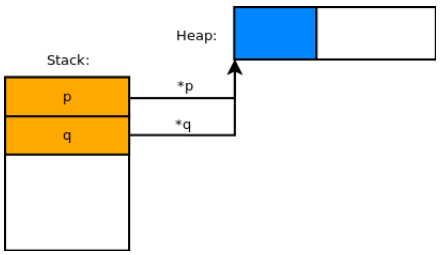
Custom Allocators Besides allocation strategies provided by operating systems and standard libraries, software sometimes relies on an ad-hoc allocator. This is particularly common in high-performance software. For example, several Internet servers, like Apache [Kor10], nginx [ngi] or Proftpd [Pro] use custom allocators based on pools. Pool-based allocators (also known as regions[CSB16] or arenas[Han90][2]) work by assigning several allocations to a similar pool, and perform deallocation on all objects of a pool at one time: it is not possible to free a single object. Custom allocators also appear in complex software, such as Adobe Reader [Li10], Flash Player [Yai11] or git[git], each of them using specific allocators. Such allocators are generally simpler than standard allocators since they are built for a specific usage. Moreover, they frequently share some properties, such as an attempt to avoid system calls, as these calls are expensive. These allocators are not well documented, and understanding their behaviors requires some reverse-engineering techniques, which can be really difficult in the case of closed source software. Automatically detecting custom allocators in a code is a challenging issue. Authors of [CSB16] propose a dynamic analysis used to detect them from a binary. While they show a good accuracy to detect allocators, understanding their specific behavior still requires a manual analysis.

2.1.3 Aliases

Understanding the heap structure generally implies analyzing pointers. One of the particularities of pointers is that two of them can refer to the same memory block; they are then called aliases. Figure 2.4a shows an example of alias. Figure 2.4b gives a view of the memory layout after the assignment `q=p;`. While performing a static analysis, a critical issue is to know if two pointers can be aliases. Such analysis is not trivial. To illustrate the difficulty of this problem, let us consider the example given in Figure 2.4c. In this case, after the condition at line 8, `q` can either be an alias to `p1` or `p2`, so each operation on `q` can refer to both pointers. The analysis thus needs to handle several possibilities, which leads to over-approximations. More generally, this problem is known to be undecidable in static analysis [Lan02].

```
1 int *p;
2 int *q;
3 p=malloc(..)
4 q=p;
```

(a) Simple alias example.



(b) Simple alias memory layout.

```
1 int *p1=..;
2 int *p2=..;
3 int *q;
4 if(cond)
5  q=p1;
6 else
7  q=p2;
8 *q=0;
```

(c) Overapproximation of aliases.

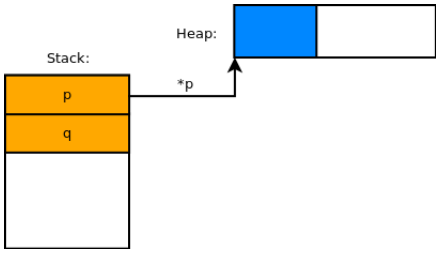
Figure 2.4: Aliases examples.

Indirect aliases Due to the nature of the heap memory, another kind of aliases can appear. They are present because of the use of previously freed memory blocks during allocations. To the best of our knowledge, the terminology for such pattern does not exist; thus we call these aliases

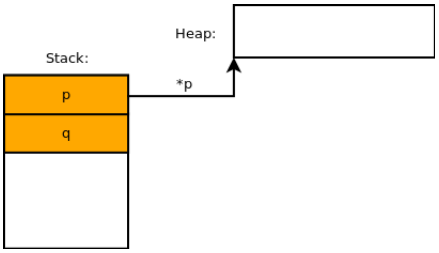
[2]This technique exists with a variety of other names.

```
1  p=malloc(..)
2  free(p);
3  q=malloc(..);
4  *q=41
5  *p=42;
6  // *q is equal 42
```

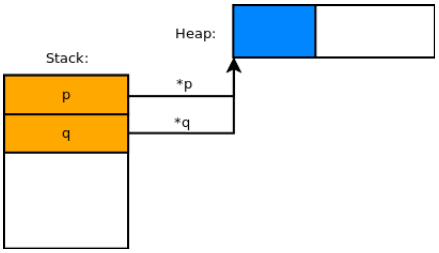
(a) Indirect aliases example.



(b) First call to malloc.



(c) Call to free.



(d) Second call to malloc.

Figure 2.5: Indirect alias examples.

`indirect aliases`. Figure 2.5 explains their behavior. Here, we make the hypothesis that the first and the second call to *malloc* give the same address (which is the most common behavior). When the chunk that was allocated by the first call to `malloc` (Figure 2.5b) is released, the memory is in the state represented in the Figure 2.5c. Because the second call to `malloc` gives the same address as the first one, `p` and `q` point to the same block, as shown in Figure 2.5d. At the binary code level, there is no direct information to differentiate the dereferencing done by `*q` and `*p`, so we cannot know if the access is valid. Notice that Figures 2.4b and 2.5d are the same, meaning that the memory is in the same state, while the executed code is different, and the access to `*p` is valid in the first case, but not in the second. Further analyses (such as data-dependency analysis) are required to distinguish these two cases during an execution.

While dynamic analysis does not suffer from the problem of direct aliases, as each pointer refers to only one address at a time, the presence of *indirect aliases* is a problem for such analysis (as we will see later in Section 2.3.2).

2.2 Use-After-Free

As we have seen, parts of the memory that are no longer in use are freed. However, after the release, pointers to the freed memory are still referencing the same blocks. Such pointers are called dangling pointers, as formalized below:

Definition 1. A dangling pointer is a pointer referencing a free block or a block reallocated to another pointer.

In Listing 2.1, after line 4, `login` is freed, yet it still points to the heap.

```
1  char *login, *password;
2  login = malloc(..);
3  ...
4  free(login); // login still points to the address returned by
       malloc
5  // login is a dangling pointer
6  password = malloc(..);
7  ..
8  printf("Login: %s\n",login); // use-after-free
```

Listing 2.1: Use-After-Free example.

A good programming practice is to assign the *NULL* value to these pointers [Sta]. In that case, such pointers are not dangling pointers anymore and future accesses trigger a *NULL* exception. Unfortunately, this practice is not always respected. Moreover, because of possible aliases (as seen in the previous section), it is not always easy for the developer to know exactly how many pointers refer to this block. Thus, removing all dangling pointers of a program is not always possible (and finding all aliases is undecidable, as seen in Section 2.1.3). Although a bad programming practice, the presence of dangling pointers is not a security issue by itself. It is the use of a dangling pointer that is considered harmful, and this issue is called Use-After-Free (Definition 2).

Definition 2. A Use-After-Free is the use of a dangling pointer.

The C11 standard refers [INC12b]:

> The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

Although undefined at the source level, the use of an object after its lifetime can happen and can have security consequences when the program is executed. In the example in Listing 2.1, the problem comes from line 8, where `login` is accessed after being freed. This example highlights one specificity of Use-After-Free exploitation: the importance of the allocation strategy. Indeed, according to the allocation strategy, the value returned by `malloc` at line 2 can potentially be the same as the value returned by `malloc` at line 6. If this is the case, `login` and `password` become *indirect aliases* (as explained in Section 2.1.3) and printing `login` prints, in fact, the value of `password`.

Use-After-Free (also referred as CWE-416 [CWEb]) is a recent vulnerability, as reflected in Figure 2.6. While it has a low CVE number in comparison to the number of other vulnera-

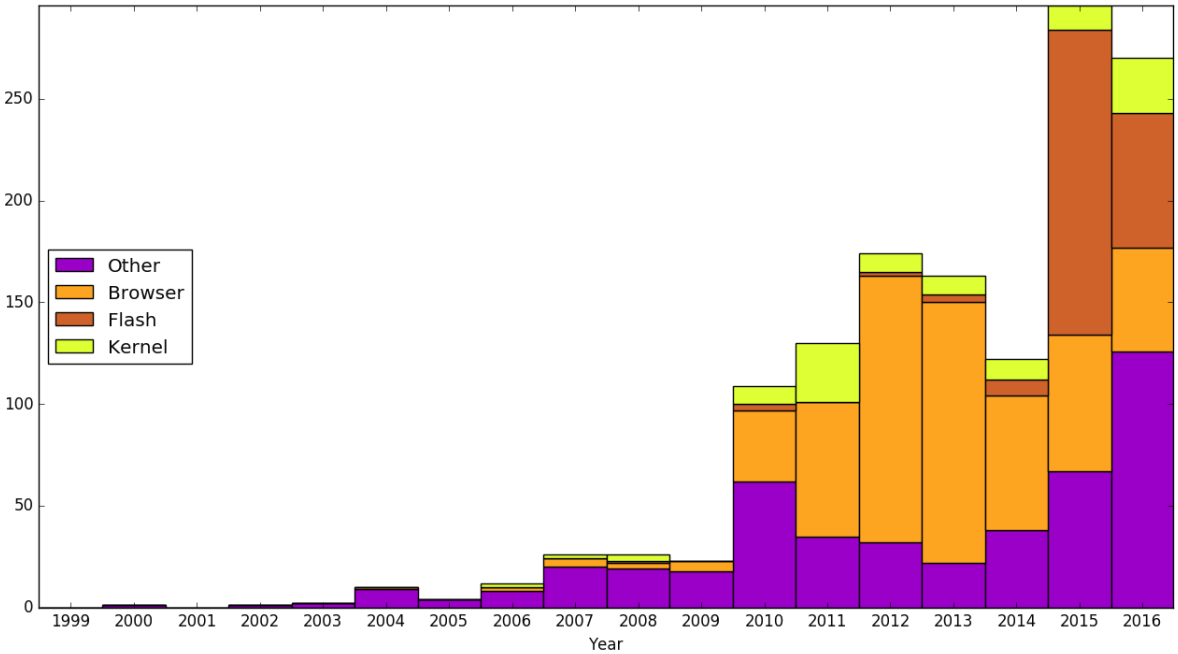


Figure 2.6: CVE: number of Use-After-Free (2016-12-28).

bilities found each year, such as buffer overflows, its danger is real. During the last Pwn2Own competition, which is a hacking contest organized during the CanSecWest conference[3], Use-After-Free were almost always present in the chain of vulnerabilities used to take down involved software (Apple OS X (CVE-2016-1804), Apple Safari (CVE-2016-1859, CVE-2016-1857,

[3] https://cansecwest.com/

CVE-2016-1856), Microsoft Windows (CVE-2016-0175, CVE-2016-0173, CVE-2016-0196), Flash (CVE-2016-1016. CVE-2016-1017), etc).

In Figure 2.6, we can notice that the first target of this vulnerability are browsers. Their complex architecture makes them more sensitive to Use-After-Free, especially because they contain several modules, such as an engine for the user interface, the network or JavaScript. Each of these modules can have a different memory management engine (Javascript uses a garbage collector, while other parts of the browser use different garbage collectors, counter references or manual management). For more information, [Gar13] presents on overview of modern browser architecture. Notice that browsers allow memory manipulation through the use of Javascript (a web page can create and destroy objects), this feature making their exploitation simpler. Other software with scripts interpretation, such as Flash, also share this property.

Double free *Double free* (CWE-415 [CWEa]) is a specific case of Use-After-Free, where a dangling pointer is used as the parameter of a call to *free*. In this dissertation, unless explicitly stated, we include directly *double free* as Use-After-Free, our techniques working similarly for both vulnerabilities.

Stack-based Use-After-Free A similar vulnerability exists on the stack: *stack-based Use-After-Free* (sometimes called *use-after-return*). This vulnerability appears when a pointer to a local variable remains active after the return of the function. It can be seen as a Use-After-Free on the stack, the local variable being freed during the return of the function and used later. This vulnerability is studied in Chapter 5.

Indirect Use-After-Free Another way of creating Use-After-Free comes from the use of uninitialized values on the stack. For example, if a function uses an uninitialized local variable as a pointer, the remaining value on the stack can be a previously freed heap pointer. It can be viewed as another way of creating aliases. This kind of vulnerability is still understudied and much harder to find. We detail it in Chapter 5.

2.2.1 Exploitation

There is no standard way of exploiting Use-After-Free. Moreover, the impact differs, whether *indirect aliases* are present or not.

```
1  typedef struct
2  {
3   void (*ptr)();
4  } st;
5
6  void nothing(){..}
7
8  void f(..){
9   st *s = malloc(sizeof(st));
10  char *buff;
11  s->ptr = &nothing;
12  s->ptr(); // call nothing
13  free(s);
14  buff = malloc(300); // buff is an indirect alias of s
15  strncpy(buff,input,300); // modify buff, so modify s
16  s->ptr(); // unexpected code execution
17 }
```

Listing 2.2: Example of code execution.

Let us consider the example in Listing 2.2. It shows the use of a dangling pointer leading to the execution of unexpected code. Indeed, if the allocation at line 14 returns the same value as the one at line 9, the copy at line 15 erases the value of `s`. Here, `s->ptr` does not point to `nothing` anymore, but to the value coming from `input`. Thus, the call leads to execute the instruction at the address given by `input`. While in most of the cases *indirect aliases* make

the vulnerability more dangerous, it is not always true. For example, for a Use-After-Free corresponding to reading a secret value that has been freed, if an indirect alias writes over this value, it is not possible to recover it.

In the case of C++ code, one of the main exploitation targets is the *virtual function* mechanism. More details can be found in [AS07], which is one of the first detailed articles explaining how to exploit a dangling pointer. Multiple references on Use-After-Free exploitation can be found in [itsa; itsb].

2.2.2 Challenges in Use-After-Free Detection

As we saw in previous sections, finding Use-After-Free requires taking into account several issues. Accurately modeling the heap is still an open problem, and allocation strategies with reallocations are generally not considered by usual techniques, which prevents the detailed study of exploitability. As seen in Section 2.1.3, Use-After-Free can imply aliases, making it even harder to find. Another particularity of this vulnerability is the distance between the events triggering the vulnerability. By comparison to buffer overflows for example, which require two events, the computation of a wrong offset and an access to it, Use-After-Free is based on three events: the allocation, the freeing and the usage. While triggering events of a buffer overflow are generally close to each others, allowing the analysis to focus on a specific part of the code (such as complex loops [Hal+13]), this is not possible for Use-After-Free. Therefore, the analysis needs to be able to take into consideration large parts of the code, raising scalability issues.

2.3 Detecting and Preventing Use-After-Free: State of the Art

In this section, we describe the state of the art techniques to detect or protect against Use-After-Free, which are either based on dynamic or static analysis. Several protections exist against vulnerabilities exploitations, *canaries* are for example used to protect against buffer overflow, and more generically, DEP / W⊕X (Data Execution Prevention) and ASLR (Address Space Layout Randomization) are widely used to protect against memory corruptions. In [Sze+14], authors offer a survey of modern protections against memory corruptions. Several strategies have been designed to detect or protect against Use-After-Free. We divide works on this subject into four categories: (i) techniques using only static analysis for the detection (both on the source and binary code levels) (ii) methods adding runtime checks (iii) allocators designed to protect against Use-After-Free, and finally (iv) protections used by particular applications, such as browsers. As we will explain in this section, browsers possess more protections against Use-After-Free than most of the classic applications.

Terminology In the following, we call **false positive** a Use-After-Free reported that cannot occur at runtime. We call **false negative** a Use-After-Free that is not detected. A **true positive** and a **true negative** are respectively a Use-After-Free correctly found and the absence of Use-After-Free correctly reported. A **root cause** is the explanation of a vulnerability.

2.3.1 Static Analyses

Static analyzers have shown their efficiency to detect vulnerabilities [EN08; Bes+10]. Several industrial tools allow to detect Use-After-Free, either on source code [HP; Graa; Mat; Cov] or binary code [Vera; Grab]. Unfortunately, we generally lack information on how they work, and it is not trivial to perform a real comparison of these tools. We can mention the Clang Static Analyzer [Cla], which offers a static symbolic execution open source framework, allowing to build checkers. A checker detecting Use-After-Free is provided; yet apart from the source code, not so many details about its properties and limitations are provided.

Several research papers describe the creation of static analyzers detecting Use-After-Free. In [Hee09], the author proposes a binary analysis based on a lightweight data flow analysis. This analysis is intraprocedural, path insensitive and does not track pointer propagations (i.e.: the creation of aliases), making it scalable on real-world examples. The analysis was applied to three

open source projects, with a low number of false positives. However, the tool was a *proof-of-concept* and seems to have not been continued. [Gio10] details a static analyzer based on the MonoREIL framework [Gooe], and performing pointer analysis to track the use of freed pointers on binary code. The analysis is interprocedural and path insensitive. While this work is close to the method we described in Part II, the report mentions only tests on small examples. Another approach to inspect binary code has been taken in [Ces13]. The author uses the decompilation of binaries to apply data-flow analysis directly on the decompiled code, detecting Use-After-Free as well as other kinds of vulnerabilities. The tool was available through a web service, but it is not longer online. More recently, [DRT15] developed a data-flow analysis focusing on binary code compiled from C++ code. It is based on heuristics to detect where C++ objects can be allocated and deleted. In [Goob], a static analysis on C++ source code is described. The tool is based on LLVM [LLV] and builds a def-use graph to perform pointer analysis. Paths in this graph are then represented as binary decision diagrams.

Limitations The lack of information, the price and the rights of use of industrial tools for academic licenses prevent us from performing a fair comparison of these techniques. While research papers allow a theoretical comparison with the static analyzer presented in Part II, none of the tools detailed in these papers are available, making an evaluation of these solutions difficult. As we saw, the detection of Use-After-Free can be performed by static analysis. However, one of the main issues in security is the number of false positives, due to the over-approximations inherent to these methods. This number can be substantial when the code to analyze was not designed following good programming practice. Lowering the number of false positives is clearly an important need. One method to check if a result is a true positive is the creation of an input triggering the vulnerability. However, this is not trivial to perform for static analysis, especially on binary code.

2.3.2 Runtime Techniques

Several runtime techniques were proposed to detect or protect against Use-After-Free.

Electric Fence [Per] and PageHeap [Mic] work by intercepting calls to the original allocator and returning instead an address pointing to a new memory page. The permissions of this page are then changed when the object is freed to forbid future accesses. [DA06] proposes a similar approach, but makes the distinction between virtual and physical pages: when a physical page has been freed it can be reused, but virtual pages cannot. Moreover, some heuristics are proposed to avoid a quick exhaustion of the address space. [DA06] allows detection with lower overhead, yet the virtual address space can still become full, and this technique cannot be adapted easily to handle custom allocators.

Metadata-based protection, introduced in [ABS94], correlates objects to metadata, holding properties such as the status of the object (e.g.: allocated or freed). The properties are propagated: the creation of an alias by an assignment `q=p` copies the metadata of `p` into that of `q`. Runtime checks on this metadata are then added to prevent memory error. Information is kept through *fat pointers*, using the property that addresses kept in pointers are aligned; their first bits are thus unused, and this space can be used for other purposes. However, this technique can break the behavior of the program in case of pointer manipulations. Authors of [XDS04] propose then to store this information outside of the object, in an expandable array. A similar approach can be found in SoftBoundCETS [Nag], an open source project. SoftBoundCETS disposes of an efficient metadata mechanism and works as a compiler transformation. It is based on the fusion of SoftBound [Nag+09] and CETS [Nag+10]. Authors of SoftBoundCETS propose another similar solution, but using a hardware implementation of pointer checking, called Watchdog [NMZ12] (and its evolution WatchdogLite [NMZ14]). A hardware implementation of such an analysis would substantially reduce the overhead. Metadata based protection is still an active area, as shown recently in [SAJ16], which uses an approach similar to SoftBoundCETS, but with better performance according to benchmarks of authors.

AddressSanitizer [Ser+12] is an open source [Gooc] memory detector for C/C++. It relies on an efficient shadow memory to track the status of the heap elements. The presence of *indirect aliases* causes AddressSanitizer to suffer from false negatives. To avoid this behavior, a custom allocator replaces the default allocator, trying to reallocate the least as possible previously freed blocks, yet it cannot avoid reallocation indefinitely. AddressSanitizer is widely used and

possesses a low overhead. It is integrated inside *gcc* and *clang* tools as an option of compilation, making this technique widely available. Notice that AddressSanitizer allows the detection of Use-After-Free on custom allocators using the *ManualPoisoning* mechanism (it is based on a source code modification). The source code is not always available, yet none of the previous runtime checkers can work without the source code. In contrast, Valgrind [NS07] is an open source framework providing dynamic binary instruction, and therefore not requiring the source code. Valgrind works on most of the popular operating systems (Linux, Solaris, Android), except for Windows. Several tools are provided within the platform; Memcheck [Val] provides a Use-After-Free checker. Similarly to AddressSanitizer, Memcheck uses a shadow memory to track the status of the chunks returned by an allocation. It intercepts calls to allocation functions and replaces them by its own custom allocator. In the same way as AddressSanitizer, Memcheck tries to reallocate the least possible number of freed chunks; yet since this is not always feasible, Memcheck can give false negatives. Similar to Valgrind, Dr.Memory [BZ11] uses a shadow memory to track Use-After-Free on binaries; according to the comparison provided by the authors, it is faster than Memcheck and works on Windows binaries. Dr.Memory is also an open source project [DrM].

Another family of methods exists, based on runtime mitigation. Instead of focusing on the detection, it offers protections against the exploitation of Use-After-Free. These mitigations have a lower overhead than classic detections techniques and thus can be executed during the normal usage of the program. DangNull [Lee+15] and FreeSentry [You15] are two mitigation techniques that work by invalidating pointers referencing to a memory area that has been freed. Access to such pointers will later crash the program, preventing an attacker from exploiting the vulnerability. FreeSentry is available as an open source tool [Cis]. Another family of mitigation focuses on protecting against specific attack vectors, such as virtual function table. For example, VTint [Zha+15a] offers protection against virtual table hijacking by forbidding this table to be written at runtime. While this protection blocks several Use-After-Free exploitations (among other types of vulnerabilities) with a low overhead, it does not protect against all exploitations of Use-After-Free.

Finally, Undangle [Cab+12] offers an *early detection* technique of dangling pointers and tracks them until they are destroyed or used. When such usage is detected, Undangle can give information on all aliases of the used pointer, and thereby help understanding the vulnerability.

Limitations Most of the previous techniques require source code, thus they cannot by applied in all contexts. Moreover, the overhead brought by most of these methods make them unusable during a normal run of the program. Thereby such solutions are more adapted during the testing phase of the program (either using unit tests or fuzzers). While the number of false positives and false negatives on a path is low (if not null), these methods require to trigger the path containing the vulnerability. Exploring all possible paths of a software is not realistic. Their capacity of detecting Use-After-Free in a program is then directly correlated to the efficiency of the underlying coverage method. Moreover, most of these methods do not allow to easily analyze custom allocators. Finally, if the technique modifies the behavior of the allocator (to avoid reallocation for example), it can change the behavior of the program, leading to infeasible paths.

2.3.3 Allocators

One way of avoiding Use-After-Free is to replace the manual management of memory by the use of a garbage collector. For example, CCured [NMW02] uses the Boehm-Demers-Weiser [Han] garbage collector and ignores calls to `free`. While this solution can be applied in some cases, a garbage collector does not provide the same performance as manual management. To keep manual management of the memory while also improving the security, specific allocators have been designed to make exploitability of heap vulnerabilities harder. In comparison to allocators described in the previous section, to limit the space overhead, the reallocation of previously freed chunks is widely used by these allocators.

Cling [Akr10] offers an allocator specifically designed to mitigate the use of dangling pointer. One of the protections provided is on the metadata, by placing it outside the freed block, using non-intrusive free lists (e.g., out-of-bands metadata) for large allocations and bitmaps for small allocations. This prevents re-allocation of the same data memory space to access the metadata.

The second protection in place is allowing reuse of chunks only for objects of the same type, preventing Use-After-Free exploitation based on the modification of fields which should not be modified (e.g., using a string object to modify the virtual function pointer of an object). The allocation strategy designed in DieHard [BZ06] and its evolution, DieHarder [NB10], mitigates exploitation using a probabilistic approach. When it allocates a new chunk, DieHarder chooses a freed chunk randomly over the set of free chunks of the appropriate size. Doing so, it is harder for an attacker to build an efficient exploit that relies on reallocations. Moreover, it fills with random data chunks that are freed. DieHarder is available as an open-source tool [Die].

Limitations These allocators have been designed to run directly as replacement of standard allocators. Overhead in the execution time and the memory consumption of such techniques can be acceptable in most cases, yet they are based on heuristics and do not ensure that a Use-After-Free cannot be exploited. Moreover, when a Use-After-Free is successfully mitigated, theses techniques do not provide information on the root cause of the vulnerability and thus do not help to fix it.

2.3.4 Browsers and Flash protections

As we saw in Figure 2.6, browsers and Flash are the principal targets of Use-After-Free. Thereby, mitigations for these specific software have been added. We detail in the following the most recent protections added by Microsoft to its browser. In July 2014, Microsoft introduced two mitigations for *Internet Explorer*: *Heap Isolation* and *Memory Protector* [TL14; Yas14]. *Heap Isolation* is basically a second heap, where specific objects are stored, while other objects are stored in the classic heap. The idea is to avoid to place internal *Internet Explorer* objects in the same location as objects that can be controlled by users (such as *string* created by Javascript). *Memory Protector* (also called *Delayed Free*) changes the mechanism of freeing objects: when an object is to be freed, it is first put in a list of objects that needs to be freed, and it is freed later. Furthermore, a check is performed, verifying that no reference to the chunk is found inside the stack or in a register. The purpose of this delay is to make it harder for an attacker to know when an object is really freed. The combination of these protections make the exploitation complex, but do not prevent all Use-After-Free attack scenarios. An example of a bypassing method, using *long-lived dangling pointer* is detailed in [DeM15]. More recently, Microsoft introduced *MemGC* [Yas15; Li15], the new memory management used in *Internet Explorer 11*, *Edge* and *Windows 10*. *MemGC* extends the concepts introduced in *Heap Isolation* and *Delayed Free* and uses *mark-and-sweep* operations to verify that an object can effectively be freed. To the best of our knowledge, this protection prevents most of the Use-After-Free to be exploited. Similar mitigations are built into other browsers. Chrome, for example, uses *PartitionAlloc* [Roh], while Firefox uses *Frame Poisoning* [OCa]. As Flash Player was one of the main targets of Use-After-Free in 2015, several mitigations have been added, and it now integrates an isolated heap [Sil16].

Limitations Doing a complete and precise report on the state of the art of browser mitigations or complex software such as Flash Player is out of the scope of this dissertation. New protections being constantly added, many technical details specific to these architectures are required to understand the mitigations fully. However, it is clear that the exploitation of Use-After-Free on such complex systems is harder than a few years ago.

Chapter 3

Illustrating the Global Approach

The present chapter gives an overview of the remainder of this dissertation. Its goal is not to detail in depth our contributions, but rather to give an insight of how the following chapters are linked together.

Outline

- We begin by briefly explaining our global approach and how it differs from state of the art in Section 3.1;
- Next, we illustrate each step of our approach through a motivating example in Sections 3.2 to 3.4;
- Finally, we discuss challenges encountered for the validation of our methods at scale in Section 3.5.

3.1 Contributions Overview

In this section, we first provide an overview of our global approach, followed by a discussion of its innovations.

3.1.1 Triggering Use-After-Free by Combining Static and Dynamic Analysis

As shown in the previous chapter, detecting Use-After-Free is not straightforward. While static analysis can help to detect complex patterns by taking into account all program paths, the number of false positives generated by such a technique is a practical limitation. Furthermore, understanding binary code is more complex than source code, making the number of false positives even higher in this context. On the other hand, by working with concrete inputs and execution traces, a dynamic analysis offers a low number of false positives. However, triggering the *right* path is difficult without specific guidance. We propose, therefore, a different approach: combining static analysis with an input generation technique, dynamic symbolic execution (DSE). We use the strength of static analysis, which is to analyze all possible paths to detect potential Use-After-Free, while DSE allows to generate an input triggering the vulnerability (also known as a *proof-of-concept*, or PoC), thus removing some false positives. The benefits of the combination are summarized in Table 3.1. Thereby, static analysis focuses on detecting the root cause while the DSE focuses on creating an input triggering the vulnerability.

	Static analysis	DSE	Static analysis + DSE
Finding Use-After-Free	+	-	+
PoC generation	-	+	+

Table 3.1: Comparison of analysis features.

Figure 3.1 illustrates the architecture of our approach. The static analyzer developed is called GUEB (Graphs of Use-After-Free Extracted from Binary). We base our DSE exploration on the BINSEC/SE framework [Dav+16a]. When a Use-After-Free is found by GUEB, it produces a slice representing several paths leading to the vulnerability. The slice is weighted and then used to guide the exploration of the DSE. If the Use-After-Free exists, the DSE can automatically create an input triggering the vulnerability. GUEB was applied to detect new vulnerabilities in several programs (see Chapter 6), and the combination with BINSEC/SE allowed to create a *proofs-of-concept* of one of these vulnerabilities.

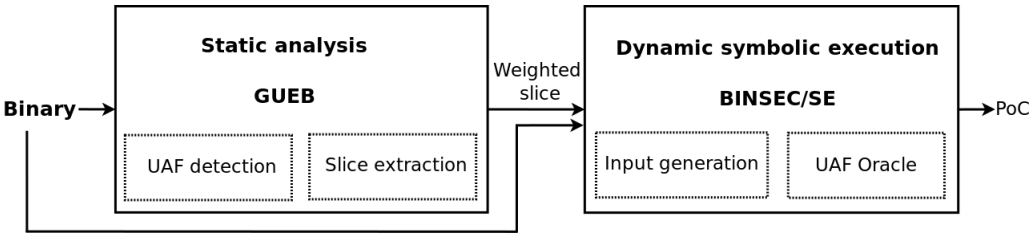


Figure 3.1: Architecture of our approach.

3.1.2 Novelties of the Approach

To the best of our knowledge, there is no system using an end-to-end approach combining static and dynamic analysis to detect and trigger Use-After-Free. In a sense, our approach is similar to [Hal+13], which is based on a static analysis aiming at locating complex loops in software, followed by a DSE guided toward these loops to find buffer overflow vulnerabilities. However, our approach goes further in the combination. First, the weighting of the slice allows guiding the DSE toward a sequence of program locations, while classic guided DSE only focuses on one specific target. Moreover, as the role of the DSE is to confirm the results of the static analysis, we can accept an unsound static analysis, making it easier to scale and to apply to binary code. We also developed an Oracle confirming the presence of a Use-After-Free in a trace with no false positives.

Another advantage of our hybrid approach is that programming constructions can be handled in a different way depending on the analysis step (e.g., static and DSE). This is the case, for instance, for loops and dynamic allocations.

Loops As described in Chapter 4, loops are complex patterns for static analysis; yet it is not mandatory to model precisely their behavior to detect Use-After-Free. However, the DSE relies on the exact number of loop iterations to generate inputs leading to the desired traces. Therefore, we do not consider loops in the same way depending on the step of the analysis:

1. During the static analysis, loops are unrolled only a few times (see Section 4.3);
2. The extracted slice keeps original loop structures (see details in Section 5.4.2);
3. During the DSE exploration, loops are iterated until an input triggering a Use-After-Free is found (Section 7.1.5).

Allocation strategies In the same way, allocation strategies are not considered similarly during the static analysis and DSE. To avoid *indirect aliases*, during the static analysis we consider a unique allocation strategy which always returns a fresh block (see Section 5.2). On the contrary, the Oracle is based on the values of the real application allocator gathered during the execution. These values are used together with a symbolic reasoning of the Oracle to confirm the vulnerability (as detailed in Section 7.3).

3.2 Thesis Motivating Example

We present in this section the motivating example that will be used in the remainder of this document. It illustrates different problems encountered by our approach.

The motivating example simulates a function triggered in case of error, but with a missing exit statement. Forgetting a call to exit or a return statement is, unfortunately, a common mistake (e.g., CVE-2013-4232, CVE-2014-8714, CVE-2014-9296, CVE-2015-7199). Figure 3.2a gives the source code of the example. Lines 20 and 27 represent potentially large parts of the program not relevant for the Use-After-Free detection. First, a memory block is allocated and assigned to `p` and `p_alias` (at lines 11 and 12). Then a file is read at line 14, and its content is placed in the buffer `buf`. If the file starts with the string `"BAD\n"`, the condition at line 16 is evaluated to `true`, and the program enters the function `bad_func`. This part of the code frees the pointer `p` (line 3). However, a call to `exit` is missing (line 4). If this path is taken, `p` and `p_alias` become dangling pointers. `p` and `p_alias` being used later (at lines 35 and 36), this could thus lead to a Use-After-Free. Notice that `p` and `p_alias` can point to another allocated block, if the condition at line 23 is evaluated to `true`, or if both conditions at line 16 and 23 are evaluated to `false`. In the former case, `p` points to a new memory block allocated at line 24, but `p_alias` is still a dangling pointer. In the latter case, both `p` and `p_alias` reference a newly allocated block, and there is no Use-After-Free.

Figure 3.2b presents the control-flow graph (CFG) associated with the motivating example. Nodes contain a high-level representation of each instruction. The two nodes illustrated in a dotted line represent large subparts of the program. The CFG will be used in the following sections to illustrate our approach. In particular, we seek paths in the CFG triggering Use-After-Free.

We split the paths in our example in four categories, according to the evaluation of the conditional nodes values and their results in term of Use-After-Free detection:

- P_1: 16 is `false` → no Use-After-Free;
- P_2: 16 is `true`, and 23 is `true` → Use-After-Free in 36;
- P_3: 16 is `true`, 23 is `false`, and 26 is `true` → Use-After-Free in 35 and 36;
- P_4: 16 is `true`, 23 is `false`, and 26 is `false` → no Use-After-Free.

Table 3.2 summarizes the possible paths leading to different outcomes. The interesting point is that paths of type P_2, P_3 and P_4 successively reach the allocation node, the free node, and the use nodes of the Use-After-Free, but only paths belonging to P_2 and P_3 trigger a vulnerability. Moreover, P_2 and P_3 trigger Use-After-Free, but only P_3 leads to a Use-After-Free in 35.

Path Type	Condition	Alloc/Free/Use node	Use-After-Free?
P_1	16:F	Y/N/N	no
P_2	16:T, 23:T	Y/Y/Y	**yes** (36)
P_3	16:T, 23:F, 26:T	Y/Y/Y	**yes** (35, 36)
P_4	16:T, 23:F, 26:F	Y/Y/Y	no

Table 3.2: Paths summary.

Root cause versus input to trigger a Use-After-Free The motivating example (Figure 3.2) illustrates the difference between the cause of the Use-After-Free and the conditions necessary to trigger it. In this case, the cause comes from the fact that after the call to `bad_func` a pointer still references the freed block, while the input triggering the vulnerability is related to the manipulation of `buf`. This distinction illustrates the different purposes that static analysis and DSE fill in our approach; the former finds the cause of Use-After-Free while the latter determines the input triggering the vulnerability.

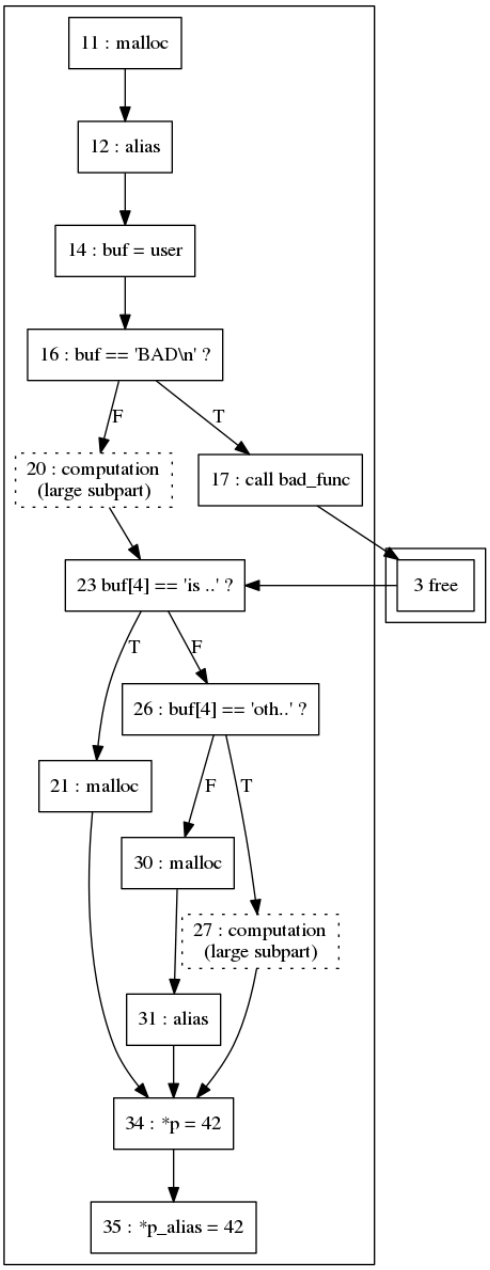
In the next section, we explain each step of our approach and how it applies to the motivating example.

3.3 Static Analysis for Detecting Use-After-Free

In this section, we give a quick overview of the static analysis that we perform, and how the results of the analysis are used. The goal here is to be able to analyze the whole binary and

```
1  void bad_func(int *p){
2   printf("This is bad, should exit !\n");
3   free(p);
4   // exit() is missing
5  }
6
7  void func(){
8   char buf[255];
9   int *p, *p_alias;
10
11  p=malloc(sizeof(int));
12  p_alias=p; // p_alias points to the same
        area as p
13
14  read(f,buf,255); // buf is user-controlled
15
16  if(strncmp(buf,"BAD\n",4) == 0){
17    bad_func(p);
18  }
19  else{
20    .. // some computation
21  }
22
23  if(strncmp(&buf[4],"is a uaf\n",9) == 0){
24    p=malloc(sizeof(int));
25  }
26  else if (strncmp(&buf[4],"other uaf\n",10)
        == 0){
27    .. // some computation
28  }
29  else{
30    p=malloc(sizeof(int));
31    p_alias=p;
32  }
33  // union of states
34
35  *p = 42 ; // is a uaf if line 16 and 26
        are true
36  *p_alias = 43 ; // is a uaf if line 16 and
        (23 or 26) are true
37 }
```

(a) Source code.



(b) Control-flow graph.

Figure 3.2: Motivating example.

to extract only interesting parts of the program. This is done in three steps: we start by performing a value analysis, then we detect possible Use-After-Free, and finally we extract slices of the control-flow graph.

Value Analysis This part of the analysis tracks the values of pointers and the allocation status of the heap blocks (e.g., allocated or freed). We describe it in details in Chapter 4.

The results of our value analysis on the motivating example is given in Figure 3.3. We represent through nodes the usage of pointers and the allocation status of the heap if it changes. In blue and green is represented the allocation status of heap blocks (blue for allocated, green for freed). In red are the dereferencing of addresses pointing to a heap block. We consider here

that each call to *malloc* returns an address to a new heap block (denoted $malloc_0$, $malloc_1$, ...). For example, the node corresponding to the line 11 is the assignment of `p` to a newly allocated heap block:

p = malloc0
malloc0: allocated

For the last two nodes of the CFG, the value analysis computes that p can be any of the three allocated blocks, while p_alias can be either $malloc0$ or $malloc2$ (in red in Figure 3.3).

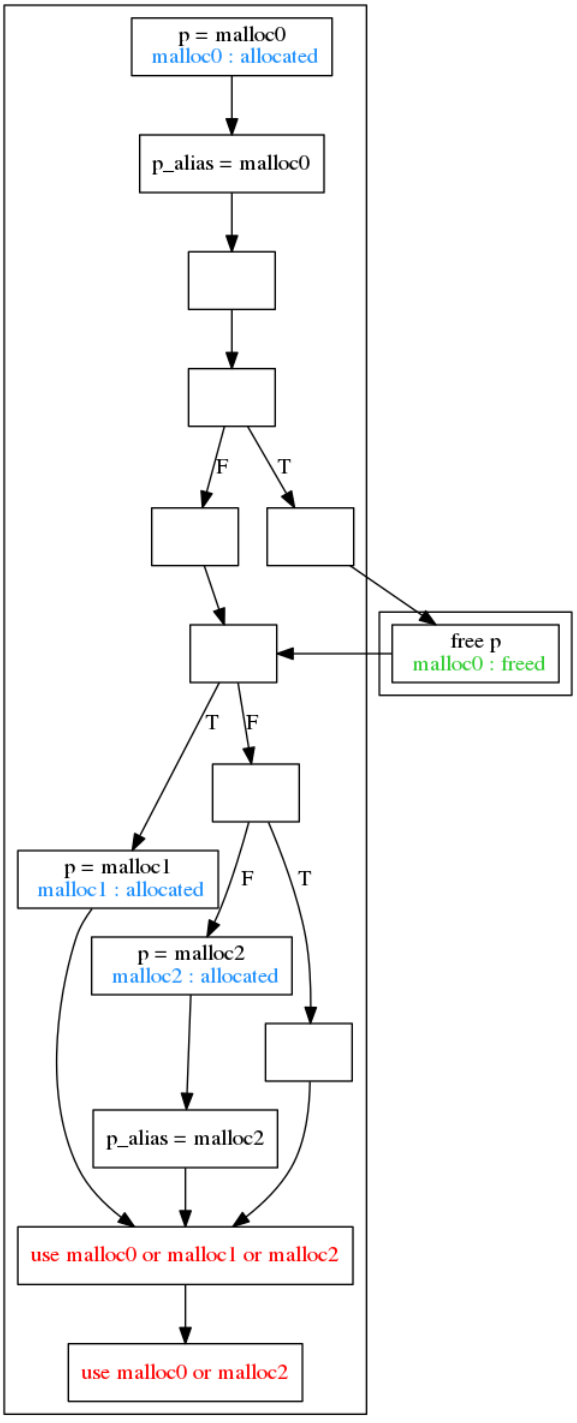
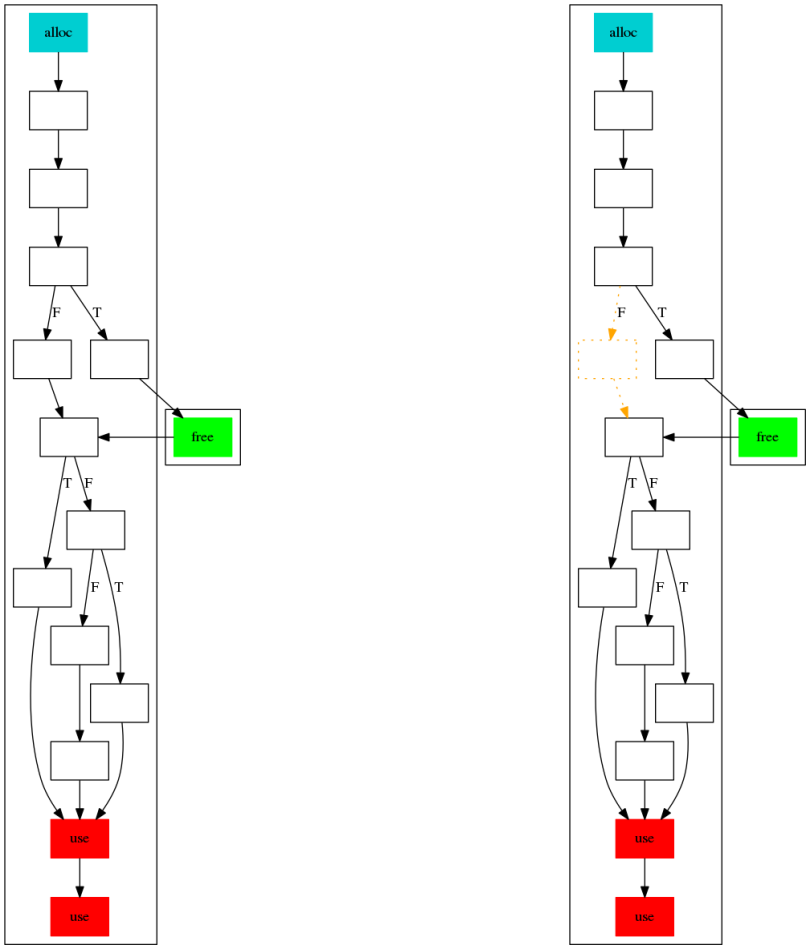


Figure 3.3: Value analysis results (blue for newly allocated blocks, green for newly freed, and red for dereferenced pointers).

Use-After-Free Detection and Slice Extraction Based on the results of the value analysis, GUEB first detects the presence of Use-After-Free and then extracts a slice representing the vulnerability. The detection corresponds to a node containing the dereferencing of a heap block, which is freed (see Chapter 5). Figure 3.4a illustrates the detection, where three specific types of nodes corresponding to the detected Use-After-Free are highlighted: the allocation node in blue, the free node in green and the use nodes in red. The extraction of the slice corresponds to all the paths reaching an allocation, a free, and a use node successively. Figure 3.4b is the slice extracted for our example; the part represented using a dotted orange line is removed. More precisely, in this example, the slicing removes line 20, which represents a potentially large part of the program.



(a) Use-After-Free Detection

(b) Slice Extraction

Figure 3.4: Use-After-Free detection.

3.4 Guided Dynamic Symbolic Execution

As introduced in Section 3.1, the role of the dynamic symbolic execution is to confirm the presence of a Use-After-Free in a slice. In addition to the exploration strategy, we need an Oracle to validate if a trace contains a Use-After-Free.

Using Weighted-Slice to Guide DSE One of the key choices for the performance of DSE is the path exploration strategy. Our heuristic weighs the slice containing the Use-After-Free (thus termed *Weighted-Slice*) and uses the weights to prioritize the path selection. The weighting allows the exploration to consider multiple targets (e.g., allocation, free and use nodes). Chapter 7

describes this process in detail.

Table 3.3 summarizes the sequence of inputs generated for our motivating example. We first try to explore the program using as input a file containing only 'A'[1]. The first condition is evaluated to `true`, and we move out of the slice (see Figure 3.5a). Without guidance, the exploration can choose to invert any condition on this path. As line 20 represents a large part of the program, with many possible conditions to invert, the probability to select the best candidate to trigger Use-After-Free can be low. However, using the slice, we can select the most appropriate condition to reach the Use-After-Free: the condition at line 16. DSE is able to invert it, thus creating a new input starting with `"BAD\n"` (as illustrated in Figure 3.5b). Both conditions 23 and 26 are still evaluated to `false` with this new input, and in this case there is no Use-After-Free, since the path belongs to P_4. DSE then creates a third input by inverting 23 or 26. If we suppose that it first inverts 23, the new input is `'BAD\nis a uaf\n'` (represented in Figure 3.5c). Since there is a Use-After-Free in the trace generated with this input, the Oracle detects it, and we now have a test case triggering the Use-After-Free. Notice that if we invert 26 first, the input generated is `'BAD\nother uaf\n'`.

Input Example	Path Type	Use-After-Free ?
"AA..A"	P_1	no
"BAD\n"	P_4	no
"BAD\nis a uaf\n"	P_2	**yes** (36)
"BAD\nother uaf\n"	P_3	**yes** (35, 36)

Table 3.3: Generated inputs summary.

Oracle Even though a path can reach all three critical nodes (i.e., the allocation, free and use nodes), it can still be a false positive. As seen in Section 3.2, paths of the type P_4 reach these three nodes but do not result in a Use-After-Free. During the concrete execution, if 16 is `true`, allocations at lines 24 and 30 can return the same address as the allocation at line 11, producing *indirect aliases*. Thereby, using `p` at line 35 and `p_alias` at line 36 may correspond to accessing the same address; yet the use of `p` is not a Use-After-Free in P_3, while in P_2 it is. On the contrary, the use of `p_alias` is a Use-After-Free in both cases. The fact that two pointers can reference the same address, but be a vulnerability only in some cases shows the complexity of the validation. To address this difficulty, we provide an Oracle that detects if a given Use-After-Free is present in a trace, with no false positives. We describe it in detail in Chapter 7.

To summarize, our combination is constituted of two steps:

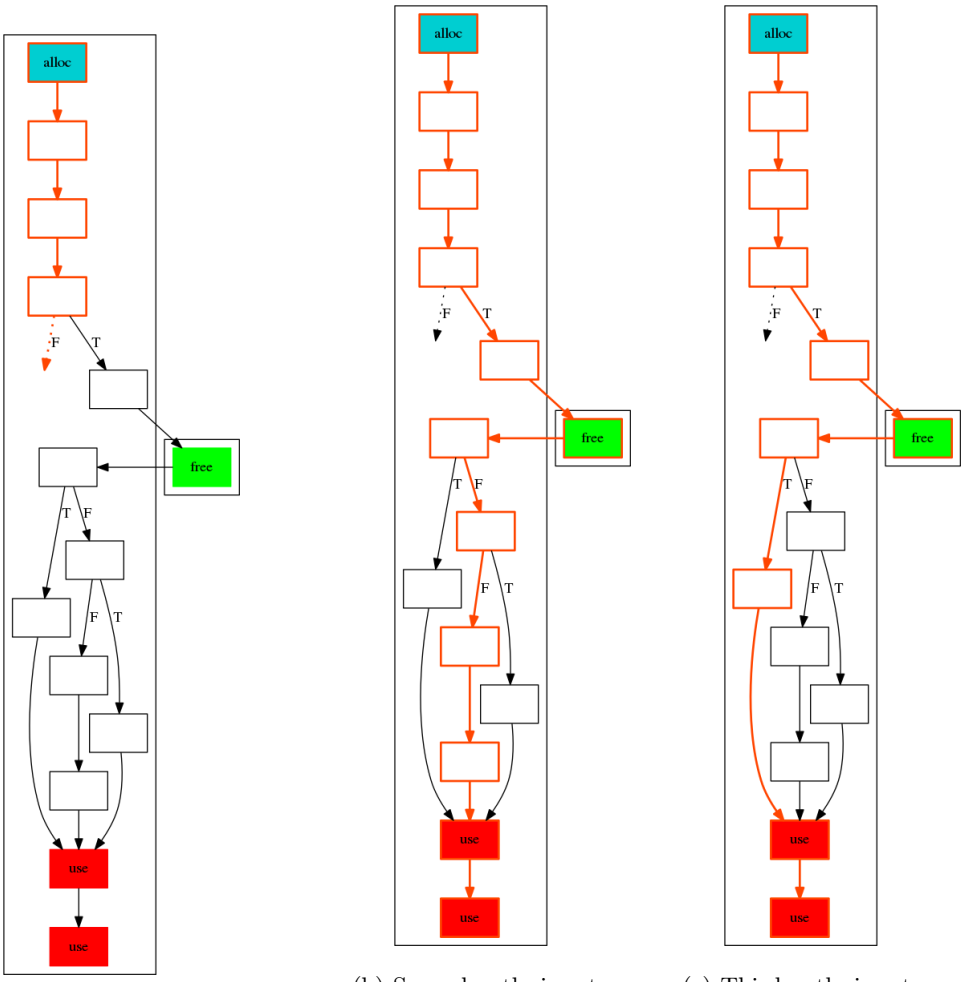
- A static analyzer to extract slice containing Use-After-Free (see Part II)
- A guided DSE to explore this slice and to create a test case of the Use-After-Free (see Part III)

3.5 Real World Analysis

We design our approach to be applicable to real-world examples; its scalability and its capacity to supply suitable results are thus two key priorities of our contribution. To fulfill these goals, we have studied several real Use-After-Free vulnerabilities (available, for example, in [AS07; Vup; itsa; itsb]). These case studies enable us to define heuristics adapted to Use-After-Free detection. In particular, our static analysis benefits from several customizations which make it scalable and efficient enough to be applied to binary code. Our method succeeded in finding several previously unknown vulnerabilities (as detailed in Chapter 6), demonstrating its efficiency. Moreover, during its design, we have used an iterative approach: the results of the tool has been used to refine and improve the heuristics.

One of the challenges encountered while developing our method was the current lack of benchmark data and measures to evaluate the performance of an analysis in detecting real Use-After-Free. Classic benchmark sets are generally developed to verify safety properties. They

[1] It is a classic seed for dynamic analysis.



(a) First path, input filled with 'A's.

(b) Second path, input 'BAD\nAAAA..'.

(c) Third path, input 'BAD\nis a uaf\n'.

Figure 3.5: DSE paths generation.

mainly rely on sophisticated, but small, pieces of code, thereby not providing metrics for computing the efficiency of a large-scale analysis while keeping a low number of false positives. For example, on the *Juliet Test Suite for C/C++ V1.2* provided by the *NIST*[2] (National Institute of Standards and Technology), the largest example of Use-After-Free is composed of 267 lines. Moreover, we have found only a small number of test cases for Use-After-Free; most of the benchmarks focus indeed on bad arithmetic computations, such as buffer or integer overflow, and rarely on the use of dangling pointers. Finally, as they are built to test source code static analyzers, such examples do not reflect difficulties encountered when applying static analysis on binary code. Testing automated vulnerability discovery tools is still an open research area. For example, [Dol+16] proposes an interesting framework which injects new vulnerabilities into a large program in order to test the capacity of a tool to find them; however, it does not consider Use-After-Free. Another interesting direction to test vulnerability detection tools can be found in [Bit]. The authors adapt binaries used during the recent DARPA Cyber Grand Challenge [Dar] to evaluate program analysis tools. Unfortunately, this benchmark was released too late to be properly integrated into this thesis.

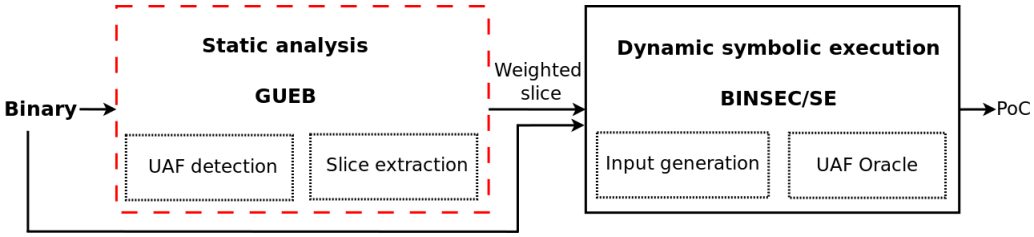
[2] https://samate.nist.gov/SRD/testsuite.php

Part II

Static Analysis

Static Analysis: Outline

This part presents the static analysis applied on binary code detecting Use-After-Free. It is the first step of our global approach, as highlighted in red in the following figure:



The presentation of our contributions to static analysis is organized in three chapters. The first two chapters provide the theoretical aspects of the analysis, while the last one focuses on the implementation and the experiments performed.

Outline and contributions In addition to the novelty of combining static analysis with DSE, we also propose several contributions on static analysis, detailed in each chapter:

- Chapter 4 provides detailed information on the application of static analysis on binary code coming from real applications;
- Chapter 5 defines several models for the heap status, as well as several heuristics adapted to Use-After-Free detection;
- Finally, Chapter 6 details the implementation of GUEB and illustrates the efficiency of our method through the analysis of real-world binary codes.

Chapter 4

Practical Binary Level Static Analysis

This chapter presents the design of a practical binary level static analysis. It defines our memory model and details several problems encountered when dealing with binary code coming from real applications. How Use-After-Free are detected is defined in the next chapter; we do not address here this issue.

Outline

- Sections 4.1 and 4.2 introduce the chapter: the former recalls some basic background about the underlying static analysis technique we use, the value set analysis (VSA), while the latter gives an overview of some requirements we need for detecting Use-After-Free on binary code.
- In Sections 4.3 and 4.4, we describe how loops and calls are handled by our analysis;
- Applying static analysis on binary code brings several problems. In Section 4.5 we explain them and present the solutions we proposed;
- The core of our static analysis, the memory model, and the VSA algorithm are formalized in Sections 4.6 and 4.7;
- Section 4.8 summarizes all the approximations made in this chapter and discusses the validity of the results;
- Section 4.9 discusses related work on binary static analysis;
- Finally, Section 4.10 concludes this chapter.

4.1 Value Set Analysis: Concepts and Bases

Static analysis is a method for analyzing a program without executing it. Value Set Analysis [BR10] (VSA) is a particular form of static analysis, well adapted to binary code. In this section, we briefly introduce the concepts necessary to understand the following sections. A complete description of our VSA is provided in Sections 4.6 and 4.7.

Intuitively, VSA aims to find all the possible values located of all memory locations at each point of the program. Starting from an initial state of the memory and at the beginning of the program (called the entry point), the effect of the first instruction on the memory state is interpreted. Then this memory state is propagated to the following instructions, and their effects are interpreted in the same way. This propagation continues until all the instructions have been visited. As the program includes control-flow structures, a join operation is used to merge the memory states of instructions that can be reached by several instructions (e.g., to merge the results of an *if then else* structure). Furthermore, due to loops and function recurrences, an instruction can be reached several times. Thus a fixed point is computed: instructions are

analyzed until the memory state converges. The interpretation of the effect of an instruction on the memory state is defined by a *transfer function*.

Consider the example in Figure 4.1a, where there are at most two possible paths, depending if the conditional jump at line 2 is taken or not. Figure 4.1b shows the effect of each instruction on the state of the memory. For this particular example, we represent all possible values at a memory location as a set of constants. At line 5, two paths are merged: either the jump at line 2 has been taken, and `eax` is 1, either it has not been taken, and `eax` is 2; thus `eax` is 1 or 2. After the addition at line 5, `eax` is 2 or 3.

```
1   mov eax,1
2   je L1
3   mov eax,2
4  L1:
5   add eax,1
```

(a) Source code

Inst	Memory state
1: mov eax,1	eax ∈ [1]
3: mov eax,2	eax ∈ [2]
5: add eax,1	eax ∈ [2,3]

(b) VSA results

Figure 4.1: Value Set Analysis example.

According to its features, a VSA tends either to precision or scalability. Trade-offs are then required to apply such technique.

4.2 Requirements for Statically Detecting Use-After-Free on Binary Code

This section introduces several observations we made related to the detection of Use-After-Free (Section 4.2.1), and the application of static analysis to real-world binary codes (Section 4.2.2). These observations will justify the choices we made in the design of our static analysis.

4.2.1 Detecting the Causes of Use-After-Free

Based on our knowledge and real studies of Use-After-Free (see Section 3.5), we have made several observations, detailed in the following paragraphs:

(i) Arithmetic computations The causes of Use-After-Free are mainly due to unexpected paths (such as the ones triggered to handle errors), rather than to an error in an arithmetic computation. Thereby, while we need to track pointers and aliases, we do not need to reason about complex arithmetic computations, such as operations on floats or complex pointers arithmetics [1]. The memory model detailed in Section 4.6 as well as the VSA algorithm described in Section 4.7 are based on this assumption; they are designed to compute simple operations as precisely as possible and to coarsly over-approximate the results of sophisticated computations.

(ii) Loops Loops introduce complexity in static analysis (see Section 4.3). Yet, in general, unrolling loops only a few times is sufficient to detect statically a Use-After-Free. Based on this observation, our analysis unrolls loops. Section 4.3 gives more details about the difficulties of handling loops at the binary level and presents our solution. To the best of our knowledge, our analysis does not miss any real Use-After-Free due to this approximation.

(iii) Scope of the analysis A feature of a static analysis is this capacity to deal with function calls. There are two strategies: either the scope of the analysis is limited to one function, either the analysis is applied to multiple functions. The former is called *intraprocedural analysis* and the latter is called *interprocedural analysis*. For Use-After-Free, an *interprocedural analysis* is generally needed, as the allocation, the free and the use sites can be located in distant locations in the code. For example, on the six unknown vulnerabilities found by GUEB (see Section 6.3), four

[1] Nevertheless, we need to treat simple pointer arithmetic operations, such as an addition of an offset to a pointer.

of them would not have been found by an analysis considering each function separately. Thereby, our analysis is *interprocedural*. More precisely, we inline function. The different solutions we adopted to do so are explained in Section 4.4.

4.2.2 Static Analysis Applied on Binary Code: Practical Observations

Analyzing real binary code involves several difficulties for static analysis:

(i) CFG recovery One of the main issues when applying static analysis at the binary level comes from the problem of disassembling the code and recovering its CFG. This problem is known to be undecidable. Although today disassemblers such as IDA [Hex], radare2 [rad], or the more recently Binary Ninja [35], provide effective solutions and are based on powerful graph reconstruction algorithms, they are not free of faults. For instance, dynamic calls generally lead to missing code of callee functions (see Section 4.5.1). As a consequence, some disassembled functions are not necessary connected to the *main* function; the analysis needs to be launched on several entry points (see Section 4.5.4). Assembly code frequently relies on indirect jumps, which can also be wrongly disassembled, leading to an incomplete control flow graph (see Section 4.5.2). All the previously mentioned constraints are taken into account in the value set analysis algorithms defined in Section 4.7.

(ii) Initial values Uninitialized values are frequently read during binary level static analysis. For example, local variables are accessed according to the initial value of the register *esp*; yet this value is not known statically. Similarly, parts of the uninitialized memory can be accessed as addresses. For example, the access of parameters of an entry point function leads to this behavior. This phenomenon is even more common in our case, as entry points for the analysis functions are not necessary the original entry point of the program. Our memory model (presented in Section 4.6) has been designed to take into account uninitialized values.

4.3 Loop Unrolling

This section presents our loop unrolling algorithm and details difficulties encountered when applying such technique to binary code.

Dealing with loops is one of the key challenges of static analysis. A classic method to ensure soundness is to iterate on the loop body until a fix-point is reached. Widening and narrowing operations [CC77] can then be used to accelerate this operation and to enforce convergence. However, using these techniques on binary codes is more difficult than on source code; for example is it challenging to find the adequate widening point or to retrieve loop conditions [Djo16; DBG16]. We choose here another classic approach, which consists in unrolling loops a given and finite number of times. Although not correct, this technique showed good results in practice.

4.3.1 Loop Structure on Binary Code

The first step for loop unrolling is to retrieve the loop structures of the targeted program. However, when applied to binary code, this operation is not straightforward since we cannot rely on the source level instructions. In addition, compiler optimizations may also modify the loop structure. Therefore we use here the classic notion of strongly connected component (SCC) on the CFG to identify loops. Loop unrolling is then reduced to SCC unrolling (dealing with nested loops inside each SCC).

4.3.2 Unrolling Algorithm

Function `Unroll` from Algorithm 1 describes how to unroll N times loops inside the set of *nodes* of a CFG. We use the Kosaraju algorithm to detect SCC [AHU83]. A SCC is selected if it contains more than one node. `find_entry` retrieves all the entry points of the SCC. A selection of the entry point is then performed using `select_entry`, which is detailed in the next paragraph. Then, edges inside the SCC reaching the entry point are removed (thus removing the cycle), using `remove_entry_edges`. To handle nested loops, we verify that there are no other

cycles inside the SCC, by recursively calling `Unroll` on it (we perform thus a depth first search exploration of SCC). Finally, the SCC is expanded by copying nodes N times (using the function `expanse_loop`), without forgetting to correctly handle newly edges between created nodes.

Algorithm 1: Loop unrolling algorithm

```
1  Algorithm Unroll(nodes,N)
2      scc_set = compute_scc(nodes)
3      foreach For scc in scc_set do
4          if |scc| == 1 then
5              do nothing
6          else
7              entry_point_set = find_entry(scc)
8              entry_point = select_entry(entry_point_set)
9              remove_entry_edges(scc,entry_point)
10             scc = unroll(scc,N)
11             expanse_loop(scc,N)
12     end
```

Reducibility of loops Loops can be classified into two categories: reducible or irreducible [Tar74; Hav97]. The difference between the two lies in the number of entry points of the loop: reducible loops have a single entry point while irreducible loops have several ones. Reducible loops correspond to well-built loops[2] while irreducible loops appear with the presence of *goto*, or due to compiler optimizations. Figure 4.2 shows an example of an irreducible loop. When unrolling loops, irreducible loops bring the problem of determining the entry node of the SCC. Most of the related work do not give details on such structures or decide to ignore them (as in [BH13]). We choose here to define a heuristic called `select_entry` which selects an entry node over a set of nodes according to this simple rule: the selected node possesses the greatest number of incoming edges, and in the case of equality, the lower address in the code. As this rule is easy to compute, its advantage is to allow handling irreducible loops with a low overhead (yet we did not study in details its real impact).

4.4 Function Inlining

This section describes how our analysis is *interprocedural*. It first presents the principle of function inlining and details the two inlining strategies we developed.

They are two types of *interprocedural analyses*. The first one is based on summaries, representing the *effects* of the function with respect to the given analysis. Functions calls are then replaced by function summaries. Each function needs to be analyzed only once, making this analysis scalable. However, building a summary in the presence of side effects coming from aliases, global variables, etc., is not straightforward. On the opposite, function inlining can be used to perform an *interprocedural analysis*. The principle of function inlining is to expand the code of the function each time the function is called.

As the analysis of aliases and side effects is necessary for Use-After-Free detection, we choose a solution based on function inlining, explained below.

4.4.1 CFG Transformation

Our VSA algorithm inlines dynamically a function when needed (see Section 4.7). This solution can be seen as a CFG transformation, performed in three steps. Let a function f calling a function g:

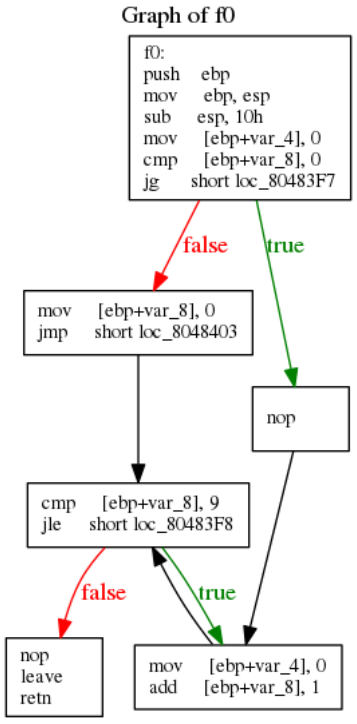
1. From the original CFG of f, the out-edge of the node corresponding to the call to g is removed;

2. This node is then connected to the entry point of a fresh copy of the CFG of g;

[2] Produced by high-level control structures, such as *while, for,...*.

```
1  void f0()
2  {
3   int i;
4   int counter=0;
5   if(i>0) goto A;
6   for(i=0;i<10;i++)
7   {
8  A:
9     counter=0;
10  }
11  return;
12 }
```

(a) Source code



(b) Control-flow graph

Figure 4.2: Irreducible loop example

3. Finally, the return nodes of g (i.e.: nodes labeled by the *ret* instruction) are connected to the caller.

Figure 4.3 illustrates an example of function inlining for the program in Listing 4.1. Here, `f3` is called twice, thus inlined twice, as well as all functions behind these calls. Notice that we remove the arc between the two successive calls to `f3` in `main`, and we add the appropriate new arcs from the return nodes.

As the same function can be called multiple times, we need to be able to distinguish it according to its context. In the following, we denote by **call string** the couple containing the function name and the address of the node calling the function. The **call stack** represents the stack containing call strings of all the callers of the function currently analyzed. In our example Listing 4.1, the two calls to `f3` leads to two different call stacks[3]: *(main,0)* , *(f3,16)* and *(main,0)* , *(f3,17)*.

Recursive calls The previous algorithm does not terminate in case of recursive calls. We do not analyze a call to a function if its call string is already present in the call stack, thereby we do not support recursion of functions. This limitation creates inconsistencies in our results (see Section 4.8). A simpler, but incomplete, approach to partially handling recursion would be to allow a call string to be present a finite number of times (this would be similar to the loop unrolling).

```
1  void f1(){
2   return ;
3  }
4
5  void f2(){
6     f1();
7     return ;
8  }
```

[3]Here, we consider as address, the line in the source code, and the address of the first node calling the entry point is 0

```
 9
10  void f3(){
11     f2();
12     return ;
13  }
14
15  void main(){
16     f3();
17     f3();
18  }
```

Listing 4.1: Illustration of inlining.

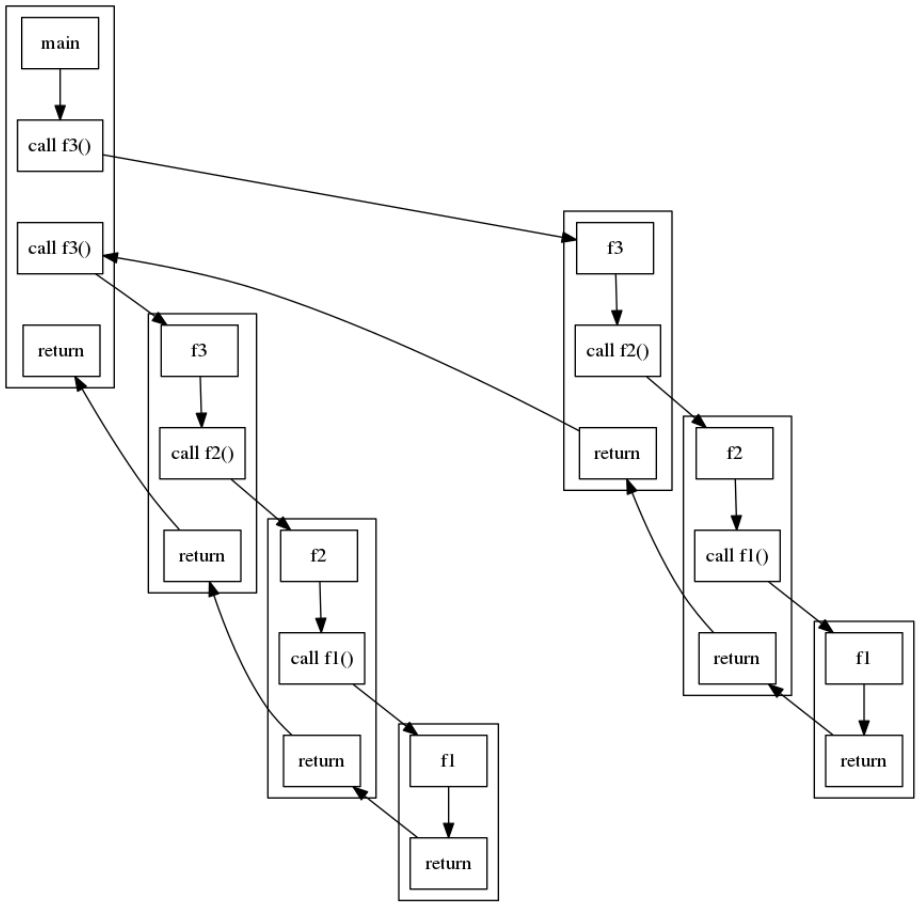


Figure 4.3: Application of inlining of the example Listing 4.1.

4.4.2 Inlining Strategies

Inlining has an impact on the scalability of the analysis. Depending on the number of calls, it is not always possible to inline all functions of a binary while still maintaining an acceptable computing time. Furthermore, combined with loop unrolling, when a call is present inside a loop, nodes of this function will be duplicated the same number of times the loop is unrolled. Consequently, we need to bound the number of functions to analyze. We propose two strategies to address this limitation, respectively called **bounded by size** and **bounded by depth**.

The first strategy consists of bounding the number of functions or instructions to analyze; we call this strategy **bounded by size**. If the number of functions is reached, following calls are not analyzed. In a similar way, if the number of instructions is reached, following instructions

are not analyzed. Figure 4.4 illustrates the results of an inlining bounded by a size of 4 functions of the example in Listing 4.1: calls after the second call to `f3` are ignored.

A second strategy consists of restricting the depth of the analysis. In this case, the application of the inlining of calls stops when the size of the call stack is larger than a fixed number; we call this strategy **bounded by depth**. Figure 4.5 illustrates the results of an inlining bounded by a depth of 2 on the example in Listing 4.1: calls to `f1` are ignored.

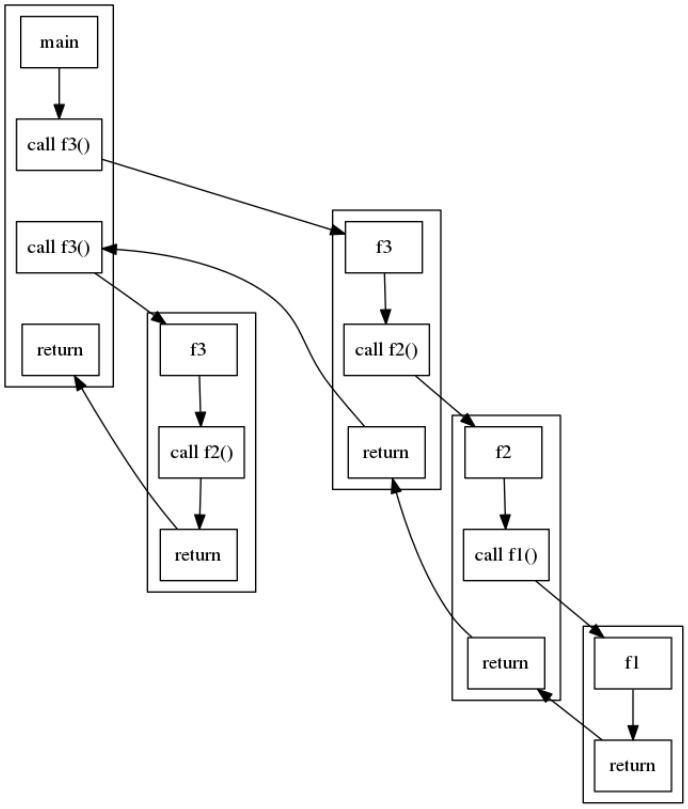


Figure 4.4: Application of an inlining bounded by a size of 4 functions.

4.5 Heuristics to Analyze Binary Code

Our value analysis is based on the control-flow graph obtained from the disassembly of the binary, which, as introduced in Section 4.2, is not free of faults. We detail in the following some challenges encountered and the heuristics we have chosen to design a binary static analyzer accurate yet applicable to real code. The validity of the results obtained from these heuristics is discussed in Section 4.8.

4.5.1 Calls to Unavailable Code

Code coming from libraries is not present in the binary code and thus cannot be analyzed. Moreover, the presence of dynamic calls leads the disassembly to not retrieve the possible destination of some calls. Thus, a disassembled binary comes with calls to unknown code.

Heuristic chosen: if the code of a call is not available, it is ignored. According to the calling convention, we over-approximate the value returned by the function (it is detailed in Section 4.7), yet we miss possible side effects of these calls, leading to inconsistencies (see Section 4.8).

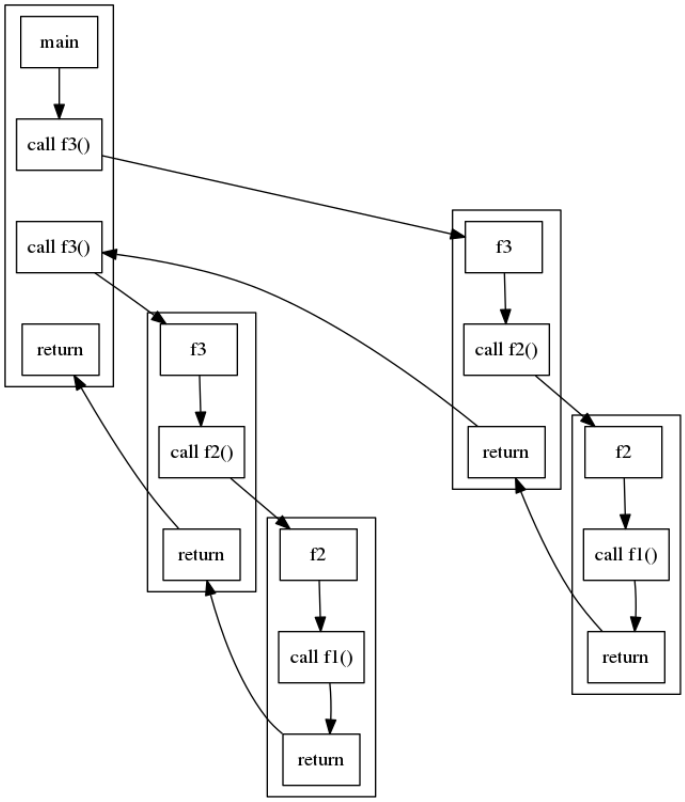


Figure 4.5: Application of an inlining bounded by a depth of 2.

4.5.2 Function Without Return Statement

Function inlining requires creating an edge between the end of the function and the node following the call (as detailed previously in Section 4.4). Return nodes of a function are nodes corresponding to the `ret` instruction. However, some functions do not possess such nodes.

For example, a function can end by jumping into the code of another function, using a `jump` instruction (generally the `ret` instruction is executed in the targeted function). Disassemblers do not support well such cases, and the recovered control-flow graph of the function ends at the `jump` instruction. While in this case, the CFG contains at least one leaf node (i.e.: containing the `jump`), this is not always the case; the CFG of a function can contain no leaf at all. For example, `without_leaf` presented in Listing 4.2 leads to a CFG without any leafs, as the function never returns.

```
1  void without_leaf(){
2   void* (*f)();
3   while(1){
4    // dispatcher that computes f
5    f();
6   }
7  }
```

Listing 4.2: Function without leaf.

Heuristic chosen: to analyze functions at best, first, we distinguish three categories of functions:

- (1) Functions with `ret` instructions (of type *normal*).
- (2) Functions without `ret` instructions, but with leaves in the CFG (called *with_leafs*);

- (3) Functions without leaves in the CFG (called *without_leaf*).

According to the category of the function, we apply different heuristics. For functions of type (1), no specific treatment is needed. For functions of type (2), the analysis returns values computed in leaves of the function (it restores the previous value of *esp* and *ebp* to maintain a correct stack frame). Functions of type (3) are analyzed, as they may contain Use-After-Free, but the effects of such functions are ignored by the caller.

4.5.3 Stack Frame Precision

The values of specific registers are crucial for the analysis: the ones determining the stack frame (e.g.: *esp* and *ebp* registers). Indeed, if the stack frame points to several locations, all access to local variables will be inaccurate, then the quality of the analysis will quickly degrade and result in being unusable.

Heuristic chosen: to avoid such behavior, our VSA verifies, on the return nodes of functions, that these registers refer to one single value. If it is not the case, their values before the call is restored, while other values computed are kept. Notice that our VSA allows local approximations, meaning that inside a function these registers can refer to multiple values.

4.5.4 Entry Points

Another particularity of static analysis applied on binary code due to the imperfections of the CFG is that not all functions are connected to the main function. For example, callbacks are generally not well handled during disassembly. Such functions are thus not analyzed if the analysis starts only from the main function. Thereby, we need to take into account several entry points for the analysis.

Heuristic chosen: we define as entry point each function that does not appear as a call target inside another function.

4.6 Memory Model

This section describes the memory model underlying our value set analysis (VSA). This memory model is inspired from [BR10] and adapted to track pointers in a scalable way.

4.6.1 Memory Model Description

Recall that VSA attaches to a memory location its possible values. In the following, a memory location is called *memLoc*, and its attached values are termed *valueSet*. We define *absEnv* (for abstract environment) as the partial function which associates, for each memory location *memLoc*, the set of possible values associated with it *valueSet*:

$$absEnv : memLoc \rightarrow valueSet$$

dom returns all memory locations on which *absEnv* is defined:

$$dom : absEnv \rightarrow P\,(memLoc)$$

We begin by formally explaining memory locations and value sets. Then we provide an example to illustrate our memory model.

Memory location

Definition 3 gives the definition of the memory location *memLoc*, using a constructor-based syntax (constructors are in bold).

Definition 3. Memory location *memLoc*

$$
\begin{aligned}
addr &= \mathbb{N} \\
offset &= \mathbb{Z} \\
chunk &= string \\
init\_reg\_name &= string \\
init\_mem\_name &= string \\
reg\_name &= string \\
memLoc &= \textbf{Globals}\ of\ addr \\
&\quad \mid \textbf{Registers}\ of\ reg\_name \\
&\quad \mid \textbf{He}\ of\ chunk \times offset \\
&\quad \mid \textbf{Init}_{\textbf{reg}}\ of\ init\_reg\_name \times offset \\
&\quad \mid \textbf{Init}_{\textbf{mem}}\ of\ init\_mem\_name \times offset \\
&\quad \mid \top_{\textbf{loc}}
\end{aligned}
$$

Memory locations are grouped into different regions. A region gathers all memory locations built with the same constructor. The division of memory locations into regions is similar to the one in [BR10], except that our regions are adapted to take into account uninitialized values. We have five types of regions:

- **Globals** contains memory locations representing direct accesses to the memory;
- **Registers** contains memory locations representing registers. We consider flags as registers.
- **He** contains memory locations representing addresses of heap objects. Each location in this region is represented by a unique identifier, called chunk, and an offset.
- Two specific regions, $\textbf{Init}_{\textbf{reg}}$ and $\textbf{Init}_{\textbf{mem}}$, contain memory locations addressed through initial values (either initial value of registers or memory), expressed by symbolic names. $\textbf{Init}_{\textbf{reg}}$ contains, for example, local variables, represented by the initial value of *esp* plus an offset. We detail these regions in Section 4.6.3.

$\top_{\textbf{loc}}$ represents any location. Offsets are integers ($\mathbb{Z}$).

Figure 4.6 illustrates examples of different memory locations over the different regions (dotted lines separate the regions). Here, each region contains two memory locations represented by their *memLoc* ($0x8000000$ for example) pointing to a set of values (illustrated by a rectangle).

Value and ValueSet

Each memory location is associated with a non-empty set of values, called *valueSet*, as formalized in Definition 4. $\top$ represents any value. Each *value* is represented by a base and a finite set of offsets. A *value* can either be a constant, a chunk of the heap, a value related to an initial register, or an initial memory. As *valueSet* is composed of a set of values; it can represent at the same time values corresponding to different bases. We consider that $absEnv(\top_{loc})$ always returns $\top$.

Definition 4. *value* and *valueSet*

$$
\begin{aligned}
base &= \textbf{Constant} \\
&\quad \mid \textbf{He}\ of\ chunk \\
&\quad \mid \textbf{Init}_{\textbf{reg}}\ of\ init\_reg\_name \\
&\quad \mid \textbf{Init}_{\textbf{mem}}\ of\ init\_mem\_name \\
value &= base \times P\,(\mathbb{N}) \\
valueSet &= P\,(value) \mid \top
\end{aligned}
$$

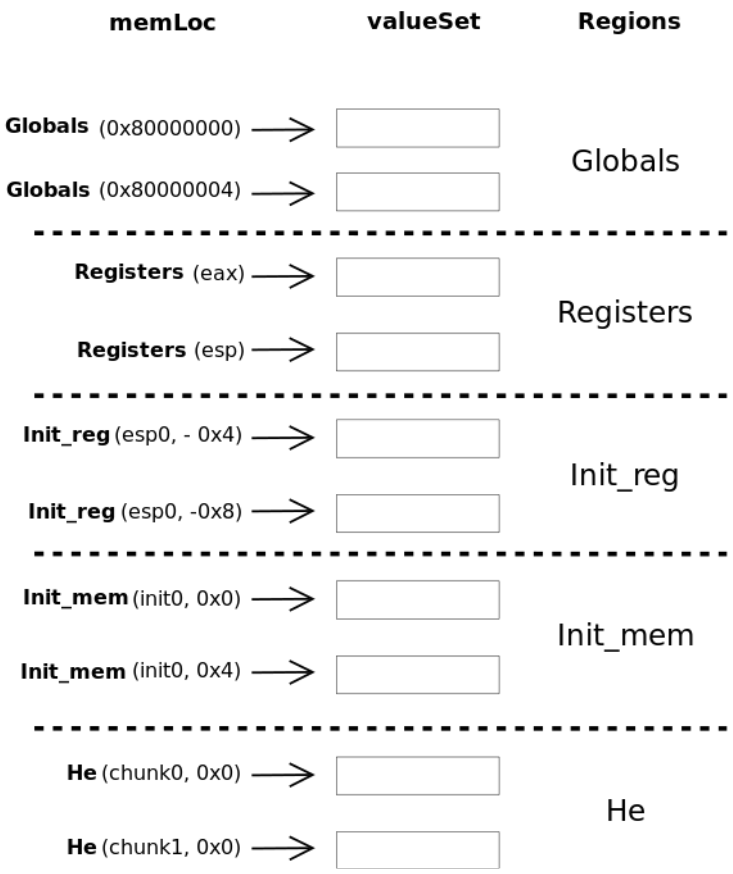


Figure 4.6: *absEnv* example

Memory Model Example

We illustrate our memory model by the example in Figure 4.7a. Here, at line 5, p can either point to 0 or to an allocated chunk. Figure 4.7b illustrates this example. The left part of the figure represents p, while the right part represents $*p$. p is accessed at the address $esp_0 - 4$, thus its address is represented by the *memLoc* $\mathbf{Init_{reg}}(esp_0, -4)$. p can either contain the constant 0, which is $(\mathbf{Constant}, (\{0\}))$, or the value returned by `malloc` represented here by $(\mathbf{He}(chunk_0), \{0\})$. We thus have:

$$absEnv(\mathbf{Init_{reg}}(esp_0, -4)) = \{(\mathbf{Constant}, \{0\}), (\mathbf{He}(chunk_0), \{0\})\}$$

If we access $*p$ later, we can either find the value located at the address 0 as a global variable, or to the values stored in the first allocated chunk, *chunk*0 (thus inside the heap region). Dereferencing a *value* corresponds to the conversion from a *valueSet* to a *memLoc*, and it is discussed in Section 4.6.6. Here, the set of offsets is $\{0\}$. Consider the following instruction located after the line 5:

```
p=p[1];
```

In this case, the *valueSet* associated to the memory location changes, and we have[4]:

$$absEnv(\mathbf{Init_{reg}}(esp_0, -4)) = \{(\mathbf{Constant}, \{4\}), (\mathbf{He}(chunk_0), \{4\})\}$$

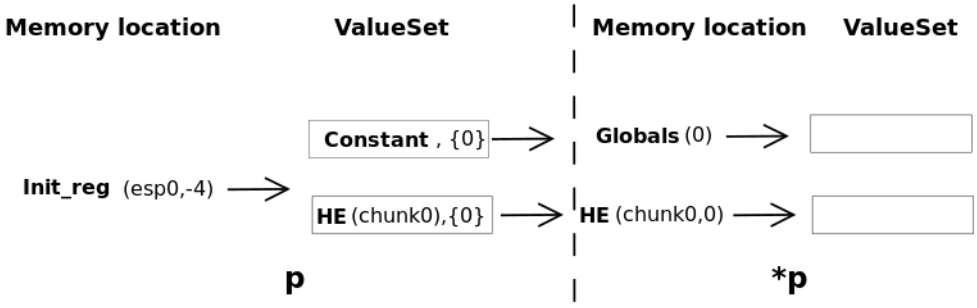
4.6.2 Overlaps between Memory Locations

In our model, and similarly to [BR10], regions are disjoint; for example, a memory location in **He** has no relation with a memory location in **Globals**. On the contrary to classic memory model, we choose to consider that each *value* attached to a *memLoc* lies in an independent

[4]We consider here `p` to be an `int` on 32 bits, thus `p[1] = p + 4`.

```
1  p=0;
2  if(cond){
3   p=malloc();
4  }
5  // merge
```

(a) Source code



(b) Memory model illustration

Figure 4.7: Pointer manipulation example.

space; there is no overlap between memory locations. For example, writing at the address a in the **Globals** region cannot change values stored at the address $a+1$ (in the same region). While this hypothesis creates inconsistencies in the results (see Section 4.8), it facilitates the analysis, as we do not need to take into account the size of the values.

4.6.3 Initial Values

Recall from Section 4.2 that uninitialized values are frequently met during the analysis. This section describes how our model deals with uninitialized values.

Initial register value A new *memLoc* can be created through the initial value of a register. For example, local variables are addressed according to the initial value of *esp*, which is undetermined. Thereby, during its initialization, our VSA assigns a symbolic initial value to all registers, corresponding to $\{\mathbf{Init_{reg}}(reg_0), \{0x0\})\}$, where *reg* is the name of the register. Notice that the initial values for registers are generally required only for *esp* and *ebp*, while other registers are written before being read (as shown in the example in Section 4.6.5).

Initial memory value We also need to handle *valueSet* on locations that are read before being written, which corresponds to reading an uninitialized part of the memory. If a memory location *loc* is accessed and not yet defined in abstract environment *abs* (i.e.: $loc \notin dom(abs)$), a new *valueSet* v is assigned to the memory location (i.e.: $abs(loc) \leftarrow v$). This value is $\{\mathbf{Init_{mem}}(n)\,\{0\}\}$, with n a fresh *init_mem_name*. We consider that all initial memory locations are pairwise disjoint; we thus do not track possible aliases between such locations (see Section 4.8 for an example).

4.6.4 Stack Frame

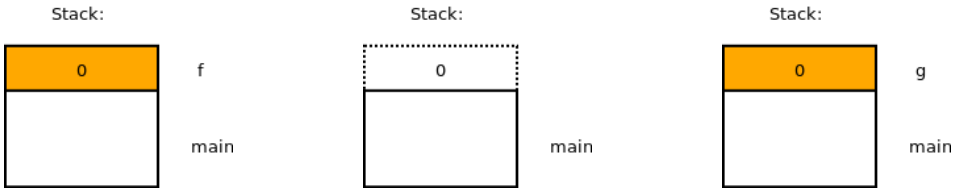
[BR10] represents each stack frame of a function as a separate region. Our model represents each local variable by a separate memory location inside the region $\mathbf{Init_{reg}}$. Having one region for all local variables allows us to represent how the same memory block is reused between successive calls, which allows matching previous values on the stack frame to uninitialized local variables. Listing 4.3 illustrates such an example. Here, `g` accesses to an uninitialized value: `b`.

```
1  void f(){
2          int a=0;
```

```
3           printf("%d\n",a);
4   }
5   void g(){
6           int b;
7           printf("%d\n",b); // b is uninitialized
8   }
9   void main(){
10          f(); // print 0
11          g(); // print 0
12  }
```

Listing 4.3: Example of uninitialized local variable.

As `f` is called before `g`, the stack frame of `g` assigns the same address for `b` as for `a`, as shown in Figure 4.8. For this reason, during the execution of `g`, `b` takes the previous value of `a`, which is 0. Our representation can take into account such an effect, while a model splitting each stack frame into a separate region would not be able to do so (as in [BR10]). Being able to follow such relations is useful to track indirect Use-After-Free, a specific variant of Use-After-Free which is described in Section 5.3. Notice that it is possible to modify our model to dedicate a separate region for each stack frame by assigning *esp* to $\{(\mathbf{Init_{reg}}(n), \{0\})\}$, with n a fresh *init_reg_name* at each function call.



(a) During `f()`. (b) After `f()`. (c) During `g()`.

Figure 4.8: Illustration of the stack of example Listing 4.3.

4.6.5 Memory Model through an Example

We illustrate in the following the results of our value set analysis when applied to the example in Listing 4.4. In particular, we illustrate the use of the region of type $Init_{reg}$ and $Init_{mem}$. Transfer functions will be defined later in Section 4.7.4, this example can be easily understandable without their complete definition.

```
1  f(){
2   int a,b; // prolog
3   a = 1;
4   if(cond)
5    a = 2;
6   b = a + 1;
7   return ;
8  }
```

Listing 4.4: Example used to detail the memory model.

Figure 4.9 shows the disassembly code produced by IDA [Hex]. The function *absEnv* associated to each instruction is provided in Table 4.1.

The effect of the instructions on *absEnv* is highlighted in red. For the clarity of the example, we group *memLoc* according to their regions. The analysis starts at the beginning of the function `f`, and we initialize *esp*, *ebp* and *eax*[5].

For example, the instruction:

```
push ebp
```

[5]Other registers are initialized, but, as they are not used in this example, we do not consider them.

Line	Code	Registers	$\mathbf{Init_{reg}}$	Globals
2	`init`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{0x0\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{reg}}(eax_0), \{0x0\})\}$		
2	`push ebp`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{reg}}(eax_0), \{0x0\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	
2	`mov ebp,esp`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{reg}}(eax_0), \{0x0\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	
2	`sub esp,0x10`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{reg}}(eax_0), \{0x0\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	
3	`mov [ebp-0x4],0x1`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{reg}}(eax_0), \{0x0\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	
4	`mov eax,ds:0x804a01c`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$
5	`mov [ebp-0x4],0x2`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x2\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$
6	`mov eax,[ebp-0x4]`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Constant}, \{0x1, 0x2\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1, 0x2\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$
6	`add eax,0x1`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Constant}, \{0x2, 0x3\})\}$	$\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1, 0x2\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$
6	`mov [ebp-0x8],eax`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x14\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(esp_0), \{-0x4\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Constant}, \{0x2, 0x3\})\}$	$\mathbf{Init_{reg}}(esp_0, -0xc) \to \{(\mathbf{Constant}, \{0x2, 0x3\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1, 0x2\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$
6	`leave`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{0x0\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Constant}, \{0x2, 0x3\})\}$	$\mathbf{Init_{reg}}(esp_0, -0xc) \to \{(\mathbf{Constant}, \{0x2, 0x3\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1, 0x2\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$
6	`ret`	$\mathbf{Registers}(esp) \to \{\mathbf{Init_{reg}}(esp_0), \{0x4\})\}$ $\mathbf{Registers}(ebp) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$ $\mathbf{Registers}(eax) \to \{\mathbf{Constant}, \{0x2, 0x3\})\}$	$\mathbf{Init_{reg}}(esp_0, -0xc) \to \{(\mathbf{Constant}, \{0x2, 0x3\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x8) \to \{(\mathbf{Constant}, \{0x1, 0x2\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x4) \to \{\mathbf{Init_{reg}}(ebp_0), \{0x0\})\}$ $\mathbf{Init_{reg}}(esp_0, 0x0) \to \{\mathbf{Init_{mem}}(init_1), \{0x0\})\}$	$\mathbf{Globals}(0x804a01c) \to \{\mathbf{Init_{mem}}(init_0), \{0x0\})\}$

Table 4.1: *absEnv* results

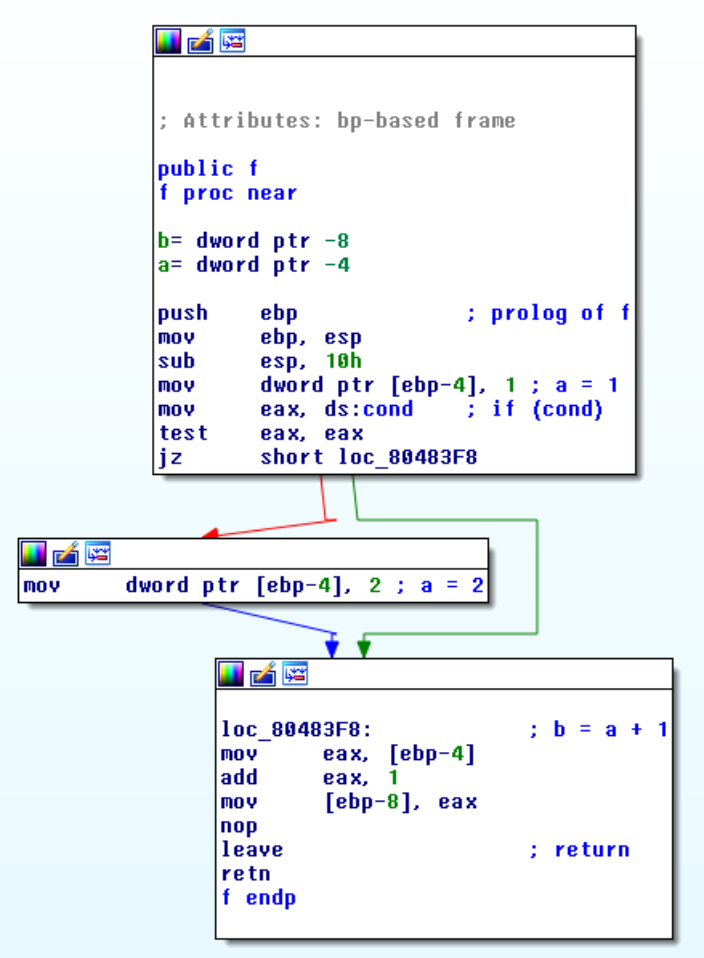


Figure 4.9: Disassembly of Listing 4.4.

has the effect of decreasing *esp* by 4 and storing the value of *ebp* in $*esp$. Thus, after this instruction, the *valueSet* associated with **Register**(*esp*) is $\{\mathbf{Init}_{\mathbf{reg}}(esp_0), \{-0x4\})\}$, and one memory location is created: $\mathbf{Init}_{\mathbf{reg}}(esp_0, -0x4)$, with the *valueSet* $\{\mathbf{Init}_{\mathbf{reg}}(ebp_0), \{0x0\})\}$.

In this example, the variable `cond` is read before being written. Thus we see here an example of initial value creation at the instruction (corresponding to the line 4):

```
mov eax,ds:0x804a01c
```

which creates an new initial value $(\mathbf{Init}_{\mathbf{mem}}(init_0), \{0x0\}))$ associated to the *memLoc* corresponding to the addr of `cond`: **Globals**$(0x804a01c)$.

Notice the creation of a second initial value due to the `ret` instruction: indeed, this instruction restores the stack, which is not initialized in this context (here, the value corresponds to the return address).

4.6.6 Converting *valueSet* into *memLoc*

A *valueSet* can be used as a memory location *memLoc*. We define *value_set_to_loc_set* as the function that transforms an element of *valueSet* to a non-empty set of locations as follows:

$$value\_set\_to\_loc\_set : valueSet \rightarrow P\,(memLoc)\,.$$

To detail this function, we first need to define *value_to_loc_set*, which takes a single *value* and converts it to a set of *memLoc*:

$$value\_to\_loc\_set : value \rightarrow P\,(memLoc)\,.$$

value_to_loc_set is defined as follows:

$$value\_to\_loc\_set(base, X) = \begin{cases} \{\mathbf{Globals}(x) \mid x \in X\} & if \quad base = \mathbf{Constant} \\ \{\mathbf{Init_{reg}}(n, x) \mid x \in X\} & if \quad base = \mathbf{Init_{reg}}(n) \\ \{\mathbf{Init_{mem}}(n, x) \mid x \in X\} & if \quad base = \mathbf{Init_{mem}}(n) \\ \{\mathbf{He}(n, x) \mid x \in X\} & if \quad base = \mathbf{He}(n) \end{cases}.$$

Then, *value_set_to_loc_set* is:

$$value\_set\_to\_loc\_set(V) = \bigcup_{v \in V} value\_to\_loc\_set(v) \ if \ V \neq \top$$
$$value\_set\_to\_loc\_set(\top) = \{\top_{loc}\}.$$

4.6.7 Union Operators

In the following, we need to perform the union of *valueSet* and *absEnv* (e.g.: during the union of paths, or for the transfer functions). We define three functions realizing these unions as follow:

$$\begin{aligned} \mathcal{N} &: P(value) \to P(value) \\ \sqcup_{vs} &: P(valueSet) \to valueSet \\ \sqcup_{abs} &: P(absEnv) \to absEnv \end{aligned}$$

We choose to keep only one similar base per *valueSet*; the normalization operator $\mathcal{N}$ is needed for this operation:

$$\mathcal{N}(\{v_1\}) = \{v_1\}$$
$$\mathcal{N}(\{(b, X), v_2, .., v_n\}) = \begin{cases} \{b, X \cup Y\} \cup \mathcal{N}(\{v_2, .., v_n\} - \{v_i\}), & if \ v_i \mid i \in 2..n, \ v_i = (b, Y) \\ \{b, X\} \cup \mathcal{N}(\{v_2, .., v_n\}) & else \end{cases}$$

$\sqcup_{vs}$ performs the union of a set of *valueSet*:

$$\sqcup_{vs}(\{vs_1, .., vs_n\}) = \begin{cases} \top & if \ \exists \top \in \{vs_1, .., vs_n\} \\ \mathcal{N}(\cup(vs_1, .., vs_n)) & else \end{cases}$$

Finally, $\sqcup_{abs}$ perform the union of *absEnv*:

$$\sqcup_{abs}\{ab_1, ..ab_n\} = \{t, \sqcup_{vs}(\cup_j(ab_j(t))) \mid j \in 1..n \wedge t \in dom(ab_j)\}$$

4.7 VSA as a Forward Data-Flow Analysis

This section gives the algorithms underlying our VSA. They implement heuristics and choices that were presented before.

4.7.1 Graph Definitions

The following definitions are used in the remainder of this document. We define a graph $G = (V, E)$ as the control-flow graph (CFG) representation of the target program. V is a set of nodes n, where each node represents an instruction of the program and E a set of directed edges. $e_{i,j}$ denotes the edge starting from the node n_i directed towards the node n_j.

For our analysis, representing a node by an instruction is not enough, as seen in Sections 4.3 and 4.4. More precisely, we want to analyze all nodes of the unrolled graph representation of the inlined program. Thereby, it is necessary to differentiate a node according to its iteration number and its call stack. In the following we use Definition 5 to represent a node:

Definition 5. A node n is represented as a triplet $ins_{addr} * it * cs$, where ins_{addr} is the address of the instruction, it the number of iterations after loop unrolling, and cs the call stack needed to reach the instruction.

From a node n, we can access to its predecessors and successors using the functions *pred* and *succ*. A leaf is a specific node without any successors. The transitive closures of *pred* and *succ* are denoted $pred^*$ and $succ^*$.

The function *is_entry_point* checks if a given node is the entry point of the current analysis (as defined in Section 4.5.4).

4.7.2 Forward Data-Flow Analysis

Our VSA is a forward data-flow analysis, meaning that it starts from the node corresponding to the selected entry point and continues by analyzing its successors.

The function *State* associates to each node n its current abstract environment as follows:

$$State : n \rightarrow absEnv$$

$State_{in}(n)$ (respectively $State_{out}(n)$) represents the abstract environment computed before entering n (respectively after exiting). A high-level representation of our VSA is given in Definition 6. The complete algorithms are given in the next section.

Definition 6. State Propagation Definition[6]

$$State_{in}(n) = \begin{cases} Init\_reg() & \text{if } is\_entry\_point(n) \\ \sqcup_{abs}\ \{State_{out}(n'),\ n' \in pred(n)\} & \text{otherwise.} \end{cases}$$

$$State_{out}(n) = \begin{cases} handle\_call(n) & \text{if } is\_call(n), \\ \mathcal{F}(n) & \text{otherwise.} \end{cases}$$

The transfer function $\mathcal{F}$, the function handling calls *handle_call*, *is_call* and *Init_reg* are defined in the next section.

Path sensitivity Our analysis is *path-insensitive*: we propagate the same *State* to all the successors of the node, regardless the path conditions. This leads to an over-approximation for the analysis, yet a *path-sensitive* analysis would require a more complex memory model and would not be as scalable.

4.7.3 Value Set Analysis Algorithm

We detail here our VSA. It is based on three main functions: $analyze\_func$, *analyze_nodes*, and *handle_call*. $analyze\_func$ is the function on which VSA is launched. *analyze_nodes* analyzes a node. *handle_call* defines how we handle calls. These three functions implement heuristics defined in Section 4.5, we give their algorithms in the following. Here, $f(x) \leftarrow y$ denotes the function defined by:

$$(f(x) \leftarrow v)(y) = \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise.} \end{cases}$$

analyze_func $analyze\_func$, defined in Algorithm 2, launches *analyze_nodes* on the entry point of the function F. It returns the *absEnv* computed by merging values of all the appropriate nodes, according to the kind of the function (as defined in Section 4.5.2). This function is based on:

- get_type_func, which determines the type of the function, according to Section 4.5.2;
- *get_ret*, which yields the return nodes of the function;
- *get_leaves*, which returns the leafs nodes of the function.

analyze_node *analyze_node* performs the analysis of a node and is defined in Algorithm 3. If all its predecessors have been analyzed, we can process the node[6]. At first, we determine the $State_{in}$ of the node, depending if it is the entry point of the analysis or not. Then we apply the corresponding transfer function to the current node n (except if the node corresponds to a call). Finally, as our VSA is a forward analysis, we apply *analyze_node* on all the successors of the node. The analysis of a node makes use of the following functions:

- $Init\_reg()$ initializes all registers to an initial value (as explained in Section 4.6.3);
- $is\_call(n)$ returns true if the node n is a call;
- $\mathcal{F}(n)$ is the transfer function and is detailed in Section 4.7.4;
- Finally, *get_root* returns the root node of a given function.

[6] As a reminder, loops being unrolled, each node requires to be analyzed only once and we do not need to compute fixpoints.

Algorithm 2: *analyze_func*

```
analyze_func(F)
    analyze_nodes(get_root(F))
    switch get_type_func(F)                                  /* Section 4.5.2 */
     do
        case normal do
            return (⊔_abs{State_out(n) | n ∈ get_ret(F)}, normal);
        end
        case with_leaves do
            return (⊔_abs{State_out(n) | n ∈ get_leaves(F)}, with_leafs);
        end
        case without_leaf do
            return (∅, without_leaf);
        end
    end
```

Algorithm 3: *analyze_node*

```
analyze_node(n)
    if is_entry_point(n) then
        State_in(n) ← Init_reg()
    else
        State_in(n) ← ⊔_abs {State_out(n') | n' ∈ pred(n)}
    end
    if is_call(n) then
        handle_call(n)
    else
        State_out(n) ← F(n)
    end
    foreach s ∈ succ(n) do
        analyze_nodes(s)
    end
```

handle_call *handle_call* is defined in Algorithm 4 and implements several heuristics of Section 4.5. It is based on the following functions:

- *ignore_call* restores the stack frame according to the calling convention and assigns $\top$ to the returned location (*eax* in *x86*);
- *check_stack_frame_consistency* checks that registers delimiting the stack frame are valid (see Section 4.5.3 for details);
- *restore_stack_frame*(a, b) restores the registers delimiting the stack frame from b to a (detailed in Sections 4.5.2 and 4.5.3);
- *get_call* returns the function F called by the node n.

4.7.4 Transfer Functions

Transfer functions describe the effects of an instruction on the proposed memory model and are provided in Algorithm 5. Classic binary analyses are applied to an intermediate representation, which contains fewer instructions than real assembly language. The assembly code is thereby translated into this intermediate representation, and the transfer function are defined on this representation. In order to simply illustrate our approach, we use a very simple intermediate representation, defined on a set of four instructions (in our implementation, we used the REIL

Algorithm 4: *handle_call*

```
handle_call(n)
    F = get_call(n)
    if Code of F is unavailable                                   /* Section 4.5.1 */
     then
        State_out(n) ← ignore_call(State_in(n))
    else
        State_ret, type_func = analyze_func(F)
        switch type_func                                          /* Section 4.5.2 */
         do
            case normal do
                if check_stack_frame_consistency(State_ret)       /* Section 4.5.3 */
                 then
                    State_out(n) ← State_ret
                else
                    State_out(n) ← restore_stack_frame(State_ret, State_in(n))
                end
            end
            case with_leaves do
                State_out(n) ← restore_stack_frame(State_ret, State_in(n))
            end
            case without_leaf do
                State_out(n) ← ignore_call(State_in(n))
            end
        end
    end
```

intermediate representation, see Chapter 6):

$$op_u\ src, dst$$
$$op_b\ src1, src2, dst$$
$$load\ src, dst$$
$$store\ dst, src$$

These instructions do not allow to convert all instructions (for example control-flow instructions can not be defined), yet they are sufficient to our purpose.

Operands (src, dst) are memory locations of type in **Globals** or **Registers**[7]. op_u and op_b respectively represent unary and binary operations (arithmetic computations, op_u^c and op_b^c are detailed in the following). *load* and *store* represent operations using the dereferencing of a value to a memory location. $load(src, dst)$ loads $*src$ into dst, while $store(dst, src)$ stores src into $*dst$.

Weak and strong memory updates Static analyses that reason with pointers generally make the distinction between two types of memory updates: *strong* and *weak updates*[BR10]. The former replaces the new value to the memory location (and thus removes the old value), while the later writes into the new memory location the union of the old and the new values. *Strong* update provides a better precision, yet when the exact location to be updated is not exactly known (e.g.: there is more than one location), *strong* update can not be used safely. In addition to classic *strong* and *weak* updates, we introduce the *ignored update*. *Ignored update* skips the store operations if the set of destination locations is $\top_{loc}$, or if it contains more than N_{loc_max} elements. This approximation makes our results inconsistent (as we will see in Section 4.8). However, it allows continuing the analysis while the location to a store operation can not be

[7] We consider here an intermediate representation using either constants or registers. Accesses to the memory are thus done through the use of registers (and potentially intermediate registers).

Algorithm 5: Transfer function $\mathcal{F}$

```
F(n)
    absEnv ← State_in(n)
    switch get_instr(n) do
        case op_u src, dst do
            absEnv(dst) ← op_u^c absEnv(src)
        end
        case op_b src1, src2, dst do
            absEnv(dst) ← absEnv(src1) op_b^c absEnv(src2)
        end
        case load src, dst do
            loc_set = value_set_to_loc_set(absEnv(src))
            absEnv(dst) ← ⊔_vs{absEnv(a) | a ∈ loc_set}
        end
        case store dst, src do
            loc_set = value_set_to_loc_set(absEnv(dst))
            if loc_set = {⊤_loc} or |loc_set| > N_loc_max then
                nothing                                   /* ignored update */
            else if loc_set = {a} then
                absEnv(a) ← absEnv(src)        /* strong update, (|loc_set| = 1)  */
            else
                absEnv(a) ← ⊔_vs{absEnv(src), absEnv(a)}, a ∈ loc_set   /* weak update
                 */
            end
        end
    end
    return absEnv
```

determined. A safe way to deal with such operations would require storing the value in all the locations, yet this would lead to unusable results. *load* operations do not need a specific handling: if the *loc_set* is $\{\top_{loc}\}$, we have $absEnv(\top_{loc}) = \top$, and thus the load operation simply puts $\top$ into the destination.

Arithmetic operations The analysis focuses only on simple binary operations. We illustrate here the computation of addition; more details about all the arithmetic operations are available in [Fei].

Table 4.2 gives the rules for an addition between two *values*: the addition is precise only if one of the two values is a constant. Otherwise, the addition returns $\top$.

$+$	**Constant**, X_1	**HE**(c_1), X_1	**Init**$_{\textbf{reg}}(n_1)$, X_1	**Init**$_{\textbf{mem}}(n_1)$, X_1
Constant, X_2	**Constant**, $X_1 +_{off} X_2$	**HE**(c_1), $X_1 +_{off} X_2$	**Init**$_{\textbf{reg}}(n_1)$, $X_1 +_{off} X_2$	**Init**$_{\textbf{mem}}(n_1)$, $X_1 +_{off} X_2$
HE(c_2), X_2	**HE**(c_2), $X_1 +_{off} X_2$	$\top$	$\top$	$\top$
Init$_{\textbf{reg}}(n_2)$, X_2	**Init**$_{\textbf{reg}}(n_2)$, $X_1 +_{off} X_2$	$\top$	$\top$	$\top$
Init$_{\textbf{mem}}(n_2)$, X_2	**Init**$_{\textbf{mem}}(n_2)$, $X_1 +_{off} X_2$	$\top$	$\top$	$\top$

Table 4.2: Addition between two *values*.

$+_{off}$ is the addition of offsets over the cartesian product of the two sets:

$$X_1 +_{off} X_2 = \{x_1 + x_2 \mid x_1 \in X_1 \text{ and } x_2 \in X_2\}$$

The addition of two *valueSet* is presented in Algorithm 6. An addition between two *valueSet* is computed precisely only if one of the two *valueSet* contains only a set of constants. Thereby, we only allow a pointer to be added to constants; adding two pointers results in $\top$.

All other arithmetic operations follow a similar approach; we precisely perform simple computations and assign the destination to $\top$ otherwise. We also use some known binary operations tricks. For example, *xor* $reg1, reg1, reg2$ results in assigning $\{(\textbf{Constant}, (\{0\})\}$ to $reg2$ regardless of the value in $reg1$.

Algorithm 6: *Add* algorithm.

add(op1,op2)
  if $op1 = \{(\textbf{Constant}, (X_1)\}$ **then**
    $\{v + (\textbf{Constant}, (X_1)) \mid v \in op2\}$ }
  else if $op2 = \{(\textbf{Constant}, (X_2)\}$ **then**
    $\{v + (\textbf{Constant}, (X_2)) \mid v \in op1\}$ }
  else
    $\top$
  end

4.8 Validity of Results

We discuss in this section the precision of our method. As we detail in the following, our VSA leads to **inconsistencies** in its results; yet, despite these approximations, the implementation that will be detailed in Chapter 6 shows real capacities for finding unknown vulnerabilities on real software, with an acceptable number of false positives.

Terminology We consider an **over-approximation** to be an analysis containing **false positives** (i.e., Use-After-Free statically detected but which do not correspond to any concrete execution of the program), while an **under-approximation** refers to analyses containing **false negatives** (i.e., Use-After-Free not detected). We denote by **inconsistent** an analysis where it is not possible to determine the validity of the results by our analysis. The consequence of **inconsistencies** is the presence of both **false positives** and **false negatives**.

4.8.1 Over-Approximations

By its nature, VSA is an over-approximation. Moreover, our VSA is path-insensitive; it thus suffers from paths containing non-possible values in its results. While a path-sensitive approach would improve the precision of the results, applying such analysis on binary is quite challenging and brings an overhead that would prevent from analyzing larger programs. *Weak updates* for the store operation (Section 4.7.4) are another cause of over-approximations.

4.8.2 Under-Approximations

As a consequence of loop unrolling, all paths are not taken into account, which can lead to missing some vulnerabilities. However, as we saw in Section 4.2, unrolling loops one or two times is enough to detect most Use-After-Free. Listing 4.5 provides an example of a complex Use-After-Free which would require a precise loop analysis or to unroll the loop at least ten iterations. While this is a counterexample, we have never met such a case in real software (as it required an access to an hardcoded offset).

```
int *buf[10];
buf[0]=malloc();
..
buf[9]=malloc();
for(i=0;i<10;i++){
 free(buf[i]);
}
buf[9][0]; // use-after-free
```

Listing 4.5: Use-After-Free dependent of loop iterations

By using our strategies for inlining *bounded by size* or *bounded by depth* (see Section 4.4), parts of the program execution are not analyzed, thus possibly leading to missing some Use-After-Free vulnerabilities. Similarly, as explained in Section 4.6.3, we do not track possible aliases between uninitialized values, thereby missing Use-After-Free dependent on such configurations. Consider example of Listing 4.6, where `f` is an entry point of the analysis. Here if the function is called with `p` and `q` being aliases, a Use-After-Free occurs. However, our model is not able to find it.

```
void f(int *p, int *q)
{
        free(p);
        q[0]=0;
}
```

Listing 4.6: Use-After-Free dependent of aliases between parameters

4.8.3 Inconsistencies

Because of potential errors in the CFG (see Section 4.2), our VSA contains inconsistencies in its results. More precisely, calls to unavailable code (see Section 4.5.1), functions without return statement (see Section 4.5.2), and stack frame imprecisions (see Section 4.5.3) lead to these inconsistencies. Similarly, the *bounded by depth* strategy for inlining or the presence of recursive calls, lead to miss side effects of function calls, which brings to the impossibility of determining the validity of the results.

Another assumption is that our model considers memory locations with no overlapping (see Section 4.6), which can also lead to invalid results. Figure 4.10a shows an example where this hypothesis results in inconsistencies. In this example, `*p` is 0x41414141 at the end. However,

```
1 char buf[4];
2 int *p;
3 buf[0]=0x41;
4 buf[1]=0x41;
5 buf[2]=0x41;
6 buf[3]=0x41;
7 p=buf;
8 printf("%x\n",*p);
```

(a) Overlap between memory location inconsistency.

```
1 int i;
2 for(i=0;i<10;i++){
3   ..
4 }
5 // i = ?
```

(b) Loop inconsistency.

Figure 4.10: Inconsistency example.

as we consider all memory locations as being disjoint, each store operation over $buf[0]$, $buf[1]$, $buf[2]$ and $buf[3]$ is represented as separate *valueSet*. Thus, when loading `*p`, the analysis only takes into account the value present in $buf[0]$ and does not consider the effects of the other three stores; `*p` is evaluated to 0x41 by our model.

Unrolling loops a fixed number of times introduces possible inconsistencies on values modified into the loop. Figure 4.10b gives such an example. Here, if we unroll the loop only twice, our model evaluates `i` to either 0, 1 or 2, while it is not possible for a path to exit the loop with such value for `i`.

Finally, if the destination address of a store operation is too complex, we ignore the operation (see *ignored update* in Section 4.7.4), which also causes the analysis to be inconsistent.

Table 4.3 summarizes all the approximations we have introduced (each impact is considered alone here).

4.9 Comparison with State of the Art

We compare in this section our work with the state of the art and we focus on practical binary static analysis. Binary static analysis has been an active research area in the past years [HM05; Son+08; Kin10; Bru+11; Sch14; MM16]. However, heuristics making these analyses practical when applied to the real world security context has, unfortunately, been less studied.

The closest work to ours is [BR10], on which our VSA is based. The static analyzer CodeSurfer/x86(CodeSurfer) is based on [BR10], showing its efficiency and applicability. In his thesis, [BR10] describes some challenges related to errors in the disassembly (in Section 3.5.2

Name	Section	Type
Path-insensitive	4.7.2	Over-approximation
Weak updates	4.7.4	Over-approximation
Unrolling (missing paths)	4.3	Under-approximation
Inlining bounded by size	4.4	Under-approximation
Inlining bounded by depth	4.4	Under-approximation
Aliases between uninitialized values	4.6.3	Under-approximation
Inlining bounded by depth	4.4	Inconsistency
Recursion	4.4	Inconsistency
Incorrect CFG	4.5	Inconsistency
No overlap between memory location	4.6	Inconsistency
Unrolling (invalid paths)	4.3	Inconsistency
Ignored updates	4.7.4	Inconsistency

Table 4.3: Summary of types of approximations for static analysis.

and 3.6 of the manuscript), partially comparable to the study we have provided in Section 4.5. For example, CodeSurfer/x86 checks the values of the stack frame registers after a call, which is similar to our solution presented in Section 4.5.3. The difference with our work comes from the fact that CodeSurfer/x86 aims to discover fine-grained information, such as the recovery of variables and their structures, while our approach focuses essentially on tracking pointers, and thus our VSA requires less precision. However, we go further in studying the challenges we have encountered. While CodeSufer/x86 raises an alert for the user if the analysis leads to imprecisions, we provide solutions to continue the analysis (such as those presented in Section 4.5.2), losing the consistency of the results, but allowing to analyze real binaries.

In [Chr12], the author describes several lessons learned when developing Veracode [Verb]. It gives several high-level advice for building a binary static analysis and focuses on design and algorithms. For example, it details guiding rules when developing suitable intermediate representation; such as the importance of a complete language features support where building the intermediate representation, or which elements to take into account.

[Ana+16] details a stack memory abstraction well adapted to binary static symbolic execution analysis. More precisely, authors focus on the effects of indirect calls and jumps on the stack frame registers. They use a symbolic stack adjustment to keep a correct valuation of the stack frame registers when an indirect call or jump could lead to an invalid computation of these registers. Using a similar adjustment when dealing with indirect calls would be an interesting improvement of our method.

4.10 Conclusion

This chapter presented the core of our value set analysis. It was driven to tackle difficulties encountered to apply VSA on real binaries. While most of the related work do not take into account these difficulties, or simply raise warnings to the user of the analysis when they appear, we decided to design heuristics dedicated to these problems. Doing so, we loose the soundness of the analysis; yet keeping tolerable inconsistencies. Results presented in Chapter 6 demonstrate that, despite the unsound analysis, our model is robust enough to be applied to real code. Our VSA is particularly suitable for the heap models detailed in the next chapter; yet all techniques presented in this chapter can be used to define other binary analyzers.

Chapter 5

Use-After-Free Detection

This chapter presents our method to detect Use-After-Free. It explains the proposed heap model, the condition to detect Use-After-Free, and the techniques we propose to improve the scalability and the usability of the results. The formalization is based on the memory model introduced in the previous chapter.

Outline

- Section 5.1 presents different models we developed for tracking the allocation status of (heap) memory chunks;
- The detection of Use-After-Free is formalized in Section 5.2;
- Section 5.3 demonstrates the strength of our model by applying it to two specifics variants of Use-After-Free: the stack-based and the indirect Use-After-Free;
- To be scalable, as well as to provide suitable results for the user, we propose several methods to represent and group similar Use-After-Free; this is discussed in Section 5.4;
- Section 5.5 discusses some related work on heap model;
- Finally, Section 5.6 concludes this chapter with a discussion of our contributions.

5.1 Heap Model with Allocation Status

In this section, we discuss how to model the status (e.g.: allocated or freed) of an heap object. In the following, this status is called *allocation status*, and we propose here two representations. The first one, called *object-based status*, is presented in Section 5.1.1. The second one, called *pointer-based status*, is presented in Section 5.1.2.

Allocation strategy Dealing with re-allocation brings the problem of *indirect aliases*, as explained in Section 2.1.3, thereby we choose to consider an allocation strategy always returning a new element. This choice does not lead to miss Use-After-Free. The heap is thus an infinite memory space and our analysis does not suffer from *indirect aliases*. This choice is common in static analysis.

5.1.1 Object-Based Status Model

To take into account Use-After-Free, a heap object has to be associated with its allocation status (allocated or freed). A natural model is to keep track of such status apart. The first version of this work [FMPHT] was built on this solution. It is based on two sets, HA and HF:

$$HA : P\,(chunk)$$
$$HF : P\,(chunk)$$

HA (respectively HF) is the set of chunks that are allocated (respectively freed). *chunk* is the identifier of a heap element, as defined in the previous Chapter. In this representation, *State*, which returns the current values computed by the VSA for a given node (as defined in Section 4.7), is extended to returns both HA and HF:

$$State : n \rightarrow absEnv \times HA \times HF$$

We call this model *object-based status*.

Figure 5.1 illustrates the allocation status of a pointer p that has been freed. The figure illustrates that the information of the allocation status is kept apart.

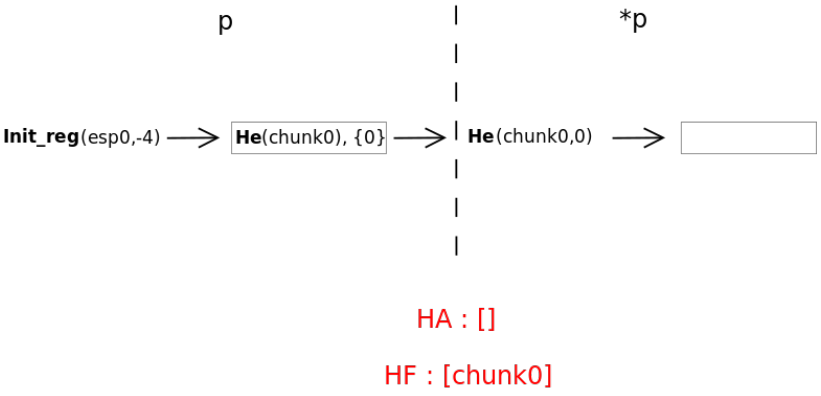


Figure 5.1: Illustration of the allocation status for a freed pointer using the *object-based* representation.

We provide the transfer functions for `malloc` and `free` in Algorithms 7 and 8. We follow in this algorithm a calling convention putting the value returned by `malloc` into the register `eax`. Freeing a *chunk* results to consider all aliases to this *chunk* as freed. We use a conservative approach when merging two paths by keeping chunks in HF rather than in HA, during the union of two *State*:

$$(HA_1, HF_1) \cup (HA_2, HF_2) \triangleq ((HA_1 \cup HA_2) - (HF_1 \cup HF_2), (HF_1 \cup HF_2))$$

Algorithm 7: Malloc using *object-based* representation

malloc(absEnv)
  $ret = (\mathbf{He}(chunk_n), \{0\})$, *with* $chunk_n$ *a fresh name*
  $HA \leftarrow HA \cup chunk_n$
  $absEnv(\mathbf{Registers}(eax)) \leftarrow \{ret\}$

Algorithm 8: Free using *object-based* representation

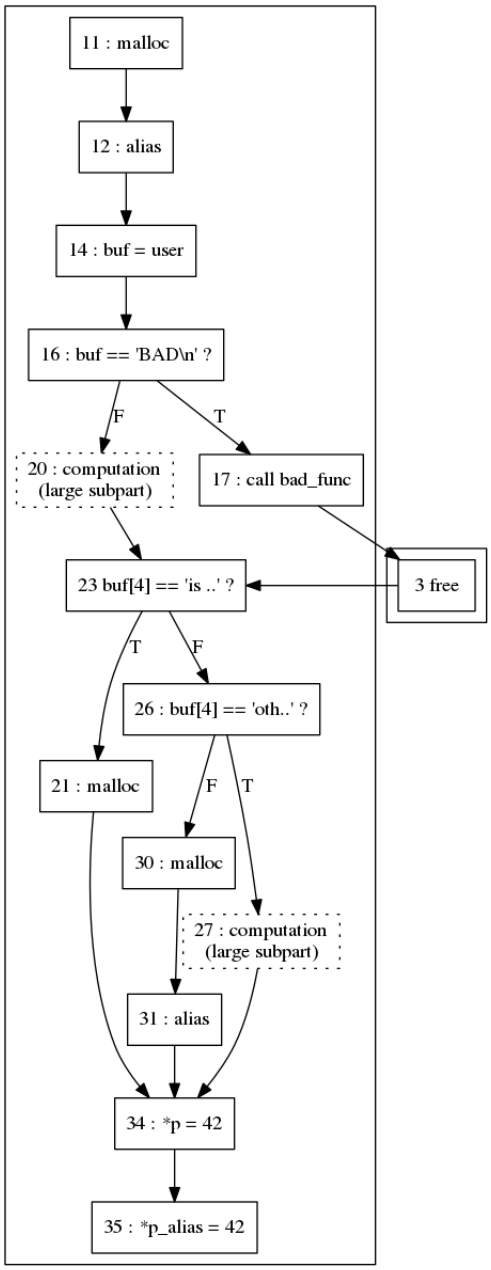
free(absEnv,p)
  foreach $(\mathbf{He}(chunk_n), N) \in absEnv(p)$ **do**
    $HA \leftarrow HA \setminus \{chunk_n\}$
    $HF \leftarrow HF \cup \{chunk_n\}$
  end

Motivating example We recall the motivating example in Figure 5.2; we are interested here in particular in the pointer manipulation.

Table 5.1 details the results of the VSA, with the *object-based* representation, on this motivating example. We assume that the two local variables `p` and `p_alias` are accessed through $\mathbf{Init_{reg}}(esp_0, -4)$ and $\mathbf{Init_{reg}}(esp_0, -8)$. After the union between memory states at line 33, `p` ($\mathbf{Init_{reg}}(esp_0, -4)$) can point to any of the allocated chunks, while `p_alias` can only reference the first allocated chunk ($chunk_0$) and the chunk allocated at line 30 ($chunk_2$). As expected, only one chunk is freed: $chunk_0$.

```
1  void bad_func(int *p){
2   printf("This is bad, should exit !\n");
3   free(p);
4   // exit() is missing
5  }
6
7  void func(){
8   char buf[255];
9   int *p, *p_alias;
10
11  p=malloc(sizeof(int));
12  p_alias=p; // p_alias points to the same
        area as p
13
14  read(f,buf,255); // buf is user-controlled
15
16  if(strncmp(buf,"BAD\n",4) == 0){
17    bad_func(p);
18  }
19  else{
20    .. // some computation
21  }
22
23  if(strncmp(&buf[4],"is a uaf\n",9) == 0){
24    p=malloc(sizeof(int));
25  }
26  else if (strncmp(&buf[4],"other uaf\n",10)
        == 0){
27    .. // some computation
28  }
29  else{
30    p=malloc(sizeof(int));
31    p_alias=p;
32  }
33  // union of states
34
35  *p = 42 ; // is a uaf if line 16 and 26
        are true
36  *p_alias = 43 ; // is a uaf if line 16 and
        (23 or 26) are true
37 }
```

(a) Source code.



(b) Control Flow Graph.

Figure 5.2: Motivating example (recall of Figure 3.2).

Limitations While this representation allows tracking the allocation status with a low overhead in space and time, it suffers from over-approximations. Let us consider the example given in Listing 5.1.

```
1  int *p=malloc(sizeof(int));
2  if(cond){
3   free(p);
4   p=malloc(sizeof(int));
5  }
6  // union of memory states
```

Listing 5.1: Example of limitations of the *object-based* representation.

Code	$\mathbf{Init_{reg}}$	Heap State
`11 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \{chunk_0\}$ $HF = \emptyset$
`12 : p_alias=p;`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$, $\mathbf{Init_{reg}}(esp_0, -0x8) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \{chunk_0\}$ $HF = \emptyset$
`3 : free(p);`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$, $\mathbf{Init_{reg}}(esp_0, -0x8) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \emptyset$ $HF = \{chunk_0\}$
`24 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_1), \{0\})\}$, $\mathbf{Init_{reg}}(esp_0, -0x8) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \{chunk_1\}$ $HF = \{chunk_0\}$
`30 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_2), \{0\})\}$, $\mathbf{Init_{reg}}(esp_0, -0x8) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \{chunk_2\}$ $HF = \{chunk_0\}$
`31 : p_alias=p;`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_2), \{0\})\}$, $\mathbf{Init_{reg}}(esp_0, -0x8) \rightarrow \{(\mathbf{He}(chunk_2), \{0\})\}$	$HA = \{chunk_2\}$ $HF = \{chunk_0\}$
`33 :// union of states`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0, \{0\}), (\mathbf{He}(chunk_1, \{0\}), , (\mathbf{He}(chunk_2, \{0\})\}$ $\mathbf{Init_{reg}}(esp_0, -0x8) \rightarrow \{(\mathbf{He}(chunk_0, \{0\}), (\mathbf{He}(chunk_2, \{0\})\}$	$HA = \{chunk_1, chunk_2\}$ $HF = \{chunk_0\}$

Table 5.1: VSA results using the *object-based* representation on the motivating example.

First, at line 1, `p` is assigned to a newly allocated chunk. At line 3 this chunk is freed, then, at line 4, `p` is assigned to a new chunk. Thereby, at line 6, `p` can either refer to the first chunk or the second. No Use-After-Free is possible is this example.

The result produced by the value analysis with an *object-based* representation is provided in Table 5.2. `p` is represented by $\mathbf{Init_{reg}}(esp_0, -0x4)$. The first chunk has $chunk_0$ for its identifier, and the second $chunk_1$. The problem comes at line 6, during the union of memory states. $\mathbf{Init_{reg}}(esp_0, -0x4)$ contains $chunk_0$ and $chunk_1$; yet $chunk_0$ is freed at line 3 and is returned by HF. As a result, future uses of `p` will be detected as possible Use-After-Free (while it is not the case).

Code	$\mathbf{Init_{reg}}$	Heap State
`1 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \{chunk_0\}$ $HF = \emptyset$
`3 : free(p);`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0), \{0\})\}$	$HA = \emptyset$ $HF = \{chunk_0\}$
`4 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_1), \{0\})\}$	$HA = \{chunk_1\}$ $HF = \{chunk_0\}$
`6 :// union of memory states`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0, \{0\}), (\mathbf{He}(chunk_1, \{0\})\}$	$HA = \{chunk_1\}$ $HF = \{chunk_0\}$

Table 5.2: VSA results using the *object-based* representation on the example Listing 5.1.

5.1.2 Pointer-Based Status Model

Based on the limitations of the previous example, we propose another representation. The idea is to attach the allocation status directly into the *value* definition. We call this model *pointer-based status*, as the status of the object is attached to the pointers referencing it. Definition 7 extends the *valueSet* to take into account this representation; we add here a *status* to the **He** constructor.

Definition 7. *valueSet* definition for the *pointer-based* representation

$$
\begin{aligned}
status &= allocated \mid freed \\
base &= \textbf{Constant} \\
&\mid \textbf{He}\ of\ chunk \times status \\
&\mid \textbf{Init}_{\textbf{reg}}\ of\ init\_reg\_name \\
&\mid \textbf{Init}_{\textbf{mem}}\ of\ init\_mem\_name \\
value &= base \times P(\mathbb{N}) \\
valueSet &= P(value) \mid \top
\end{aligned}
$$

Figure 5.3 represents the modeling of a pointer p that has been freed using pointer-based status representation, as opposed to Figure 5.1 previously introduced.

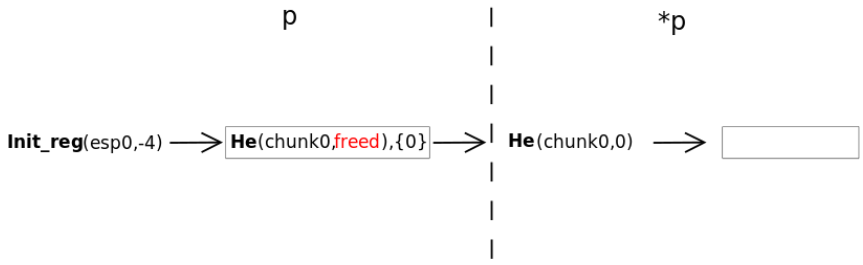


Figure 5.3: Illustration of the allocation status for a freed pointer using the *pointer-based* representation.

Algorithms 9 and 10 detail the transfer function of `malloc` and `free` in this representation. In particular, a call to `free` implies that all pointers referencing the freed chunk have to be marked as freed; thus we need to iterate over all possible values of the abstract environment. $free\_chunk$ frees a *value* if it has for constructor the *chunk* to be freed. Similar to the *object-based* representation, if a chunk is both allocated and freed during the union of states, we consider it as freed.

Algorithm 9: Malloc using *pointer-based* representation.

$malloc(absEnv)$
  $ret = (\textbf{He}(chunk_n, allocated), \{0\})$ *with* $chunk_n$ *a fresh name*
  $absEnv(\textbf{Registers}(eax)) \leftarrow \{ret\}$

Algorithm 10: Free using *pointer-based* representation.

$free\_chunk(v,chunk_n)$
  if $v = (\textbf{He}(chunk_n, s), X)$ **then**
    return $\textbf{He}(chunk_n, freed), X)$
  else
    return v

$free(absEnv,p)$
  foreach $(\textbf{He}(chunk_n, s), N) \in absEnv(p)$ **do**
    foreach $(loc, vals) \in absEnv$ **do**
      $absEnv(loc) \leftarrow \{free\_chunk(v, chunk_n) \mid v \in vals\}$;
    end
  end

The result of the value analysis using this representation on the example in Listing 5.1 is given in Table 5.3. As one can notice, the dangling pointer to $chunk_0$ disappears at line 4, and

there is no more reference to the freed chunk. During the union of the memory states, `p` points either to $chunk_0$ (allocated) or $chunk_1$ (allocated). Then no false Use-After-Free induced by a future use of `p` will be detected.

Code	$\mathbf{Init_{reg}}$
`1 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0, allocated), \{0\})\}$
`3 : free(p);`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_0, freed), \{0\})\}$
`4 : p=malloc(sizeof(int));`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \{(\mathbf{He}(chunk_1, allocated), \{0\})\}$
`6 :// union of states`	$\mathbf{Init_{reg}}(esp_0, -0x4) \rightarrow \begin{array}{l}\{(\mathbf{He}(chunk_0, allocated), \{0\}),\\ (\mathbf{He}(chunk_1, allocated), \{0\})\}\end{array}$

Table 5.3: VSA results using the *pointer-based* representation on the example Listing 5.1.

Limitations This representation is more precise than the previous one but it introduces an overhead: all pointers referencing the chunk to be freed have to be marked as freed. Yet, our benchmarks (Chapter 6) show that this overhead is negligible, mainly because there are generally not so many calls to `free`.

5.2 Use-After-Free Detection

In this section, we formalize the detection of Use-After-Free. From the CFG and the values computed by the VSA (using one of the two allocation status previously defined), our goal is to detect the dereferencing of a *chunk* that has been freed.

The formalization of the detection is organized in three steps: (i) obtaining which *chunk* are freed from a given a set of values, (ii) detecting if a *chunk* is freed and used in a given node, and finally (iii) the generalization of this detection over all the nodes of the CFG.

(i) Obtaining freed chunks
We define *get_free*, the function that returns the freed chunks among a *valueSet*:

$$get\_free : valueSet \rightarrow P\,(chunk)$$

Algorithm 11 gives the definition of get_free for object-based representation, while Algorithm 12 is its counterpart for pointer-based representation.

Algorithm 11: get_free for `object-based`.

$get\_free(vals)$
  return $\{chunk_x \mid (\mathbf{He}(chunk_x), N) \in vals \text{ and } chunk_x \in HF\}$

Algorithm 12: get_free for `pointer-based`.

$get\_free(vals)$
  return $\{\ chunk_x \mid (\mathbf{He}(chunk_x, freed), N) \in vals\}$

On our motivating example (Figure 5.2), from the results of the VSA, using *object-based* representation (Table 5.1), get_free on the *valueSet* associate to p at line 35, results in:

$$get\_free(\{(\mathbf{He}(chunk_0, \{0\}), (\mathbf{He}(chunk_1, \{0\}), \mathbf{He}(chunk_2, \{0\})\}) = \{chunk_0\}$$

Notice that get_free using *pointer-based* representation would, here, returns the same result.

(ii) Detecting Use-After-Free on a node
Then we can define get_uaf_set returning the set of possible chunks that are freed and then used for a given node:

$$get\_uaf\_set : n \rightarrow P\,(chunk)$$

Algorithm 13: *get_uaf_set*.

```
get_uaf_set(n)
    absEnv ← State_out(n)
    switch get_instr(n) do
        case load src, dst do
            return get_free(absEnv(src))
        end
        case store src, dst do
            return get_free(absEnv(dst))
        end
        otherwise do
            return ∅
        end
    end
```

Algorithm 13 defines *get_uaf_set* on the intermediate language described in Section 4.7.4. Recall that this language possesses two instructions to dereference a value: load and store.

get_uaf_set apply to the line 35 of our motivating example results in:

$$get\_uaf\_set(35) = \{chunk_0\}$$

(iii) Detecting all Use-After-Free of a CFG
We now define the set of all possible Use-After-Free as follows:

Definition 8. Use-After-Free characterisation
$UafSet = \{(n, chunk) \mid chunk \in get\_uaf\_set(n))\}$

On our motivating example it results in: $UafSet = \{(35, chunk_0), (36, chunk_0)\}$.

5.2.1 Free Applied to Uninitialized Values

Because of uninitialized memory, the function *free* can be applied to *values* of type $\mathbf{Init_{mem}}$. To not miss possible Use-After-Free, when such pattern occurs, the analysis updates the constructor of the *value* to **He** (with a fresh *chunk* as identifier).

Consider the example Listing 5.2, where *callback* is an entry point for the analysis. Our VSA determines that `p` points to one uninitialized value (with $\mathbf{Init_{mem}}$ as the constructor). As it is used as the parameter of *free*, we make the hypothesis that p points in fact to a heap element and promote its constructor to **He**. Thus we can detect the presence of the Use-After-Free with Definition 8.

```
1  void callback(int *p)
2  {
3          free(p);
4          *p=0;
5  }
```

Listing 5.2: Free uninitialized value example.

5.3 Use-After-Free Variants

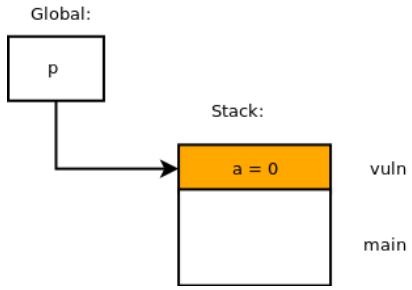
The models presented previously allow to detect classic heap Use-After-Free. However, as mentioned in Section 2.2, other types of Use-After-Free exist. In this section, we describe in detail Use-After-Free occurring on the stack, as well as how uninitialized values can produce Use-After-Free. Then we explain how our model can detect them. These variants are less studied, and most of state of the art tools do not take them into consideration. A shared particularity of these Use-After-Free if that they require a precise layout of the stack model to be detected; our VSA is thereby well adapted to detect them.

5.3.1 Stack-Based Use-After-Free

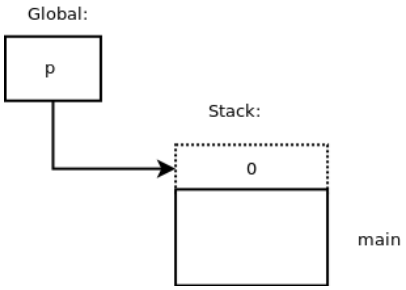
A first variant is Use-After-Free on stack variables (also called *use-after-return*). The allocation of local variables is made during the prolog of the function, while these variables are freed when the function returns. Figure 5.4a shows an example of stack-based Use-After-Free. Here, the

```
1  int *p;
2
3  void vuln()
4  {
5          int a=0;
6          p=&a;
7  }
8
9  int f0()
10 {
11         int b=0;
12         *p=42;
13         return b;
14 }
15
16 void main()
17 {
18         vuln();
19         printf("f0=%d\n",f0());
20 }
```

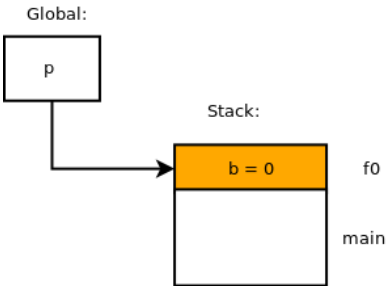
(a) Source code.



(b) In vuln()



(c) After vuln()



(d) In f0()

Figure 5.4: Stack-based Use-After-Free example.

address of the local variable `a` in the function `vuln` is kept through the global pointer `p`, as represented in Figure 5.4b. After the return of `vuln`, `a` is freed. An access to it is now invalid, but `p` still points to the memory area previously used by `a`, as depicted in Figure 5.4c. If we consider that the stack layout for the call to `f0` provides the same memory address to `b` as to `a` (Figure 5.4d), an access to `b`, or to `p` is an access to the same memory region, yet the later is a stack-based Use-After-Free.

To detect such situations, we need to track the status of the stack variables. We propose here a similar approach to the *pointer-based* representation, introduced in the previous section. When a value points to an element in the stack, we associate an allocation *status* to it. To determine which values are pointing to local variables, we consider a new constructor of *memLoc* in *AbsEnv*:

$$\textbf{Stack}\ of\ name\_stack \times offset$$

with $name\_stack = \{esp_0 \mid ebp_0\}$. The definition of *valueSet* for the detection of stack-based Use-After-Free is extended in Definition 9.

Definition 9. *valueSet* definition for stack-based Use-After-Free

$$
\begin{aligned}
\mathit{name\_stack} &= \mathit{esp}_0 \mid \mathit{ebp}_0 \\
\mathit{status} &= \mathit{allocated} \mid \mathit{freed} \\
\mathit{base} &= \textbf{Constant} \\
&\quad \mid \textbf{He}\ \mathit{of}\ \mathit{chunk} \times \mathit{status} \\
&\quad \mid \textbf{Init}_{\textbf{reg}}\ \mathit{of}\ \mathit{init\_reg\_name} \\
&\quad \mid \textbf{Init}_{\textbf{mem}}\ \mathit{of}\ \mathit{init\_mem\_name} \\
&\quad \mid \textbf{Stack}\ \mathit{of}\ \mathit{name\_stack} \times \mathit{status} \\
\mathit{value} &= \mathit{base} \times P\,(\mathbb{N}) \\
\mathit{valueSet} &= P\,(\mathit{value}) \mid \top
\end{aligned}
$$

Upon the initialization of the *State* to the entry point of the analysis, we assign *esp* to $\{(\textbf{Stack}(\mathit{esp}_0, \mathit{allocated}), \{0\})\}$ and *ebp* to $\{(\textbf{Stack}(\mathit{ebp}_0, \mathit{allocated}), \{0\})\}$ (see $\mathit{Init\_reg}()$ in Section 4.7.3). If an access is performed through *esp* or *ebp*, it will be considered as a stack element. If it is related to an access through another register, it will be tracked using the $\mathit{Init}_{\mathit{reg}}$ region.

After the return of a function, values pointing to an address lower than the current stack frame are freed[1]. Thus, if a pointer references a freed local variable, a future access to it is detected as Use-After-Free.

$\mathit{get\_free}$ for this representation is given in Algorithm 14. Here, *create_chunk* returns a fresh *chunk* created from a *value*. Based on Algorithm 14, Definition 8 (Section 5.2) remains correct to detect stack-based Use-After-Free.

Algorithm 14: $\mathit{get\_free}$ for `stack-based` Use-After-Free.

$\mathit{get\_free}(\mathit{vals})$
  return $\{\ (\mathit{chunk}_x \mid (\textbf{He}(\mathit{chunk}_x, \mathit{freed}), N) \in \mathit{vals}) \cup$
  $\{\mathit{create\_chunk}(\textbf{Stack}(n, \mathit{freed}), N) \mid \textbf{Stack}(n, \mathit{freed}), N) \in \mathit{vals}\}\ \}$

In our example, after the call to `vuln`, we assign the status of the pointer `p` to `freed`, as it points to a value lower than the current stack frame; thus the access at line 12 is detected as a stack-based Use-After-Free.

5.3.2 Indirect Use-After-Free

Another particular type of Use-After-Free can occur due to an uninitialized pointer. Figure 5.5 gives such example.

First, the function `func` is called. A chunk is allocated to the local variable `p`, as illustrates in Figure 5.5b, then freed. The interesting point is that `p` is a dangling pointer during the return of the function; thus, when the function returns, the address of the chunk is still located in the stack (see Figure 5.5c). When `vuln` is called, two local variables are allocated on the stack, `p` and `q`. If we make the hypothesis that `p` from `vuln` is allocated at the same address as `p` from `func`, `p` from `vuln` points to the freed chunk. If the allocation strategy gives to `q` the same address as the previously freed chunk (which is the common case), `p` and `q` point to the same area. As `p` of `vuln` is not assigned to any value; it has the same value than `p` of `func`, which points to the heap chunk. Then the modification of `*p` leads to also changing `*q`, as shown in Figure 5.5d.

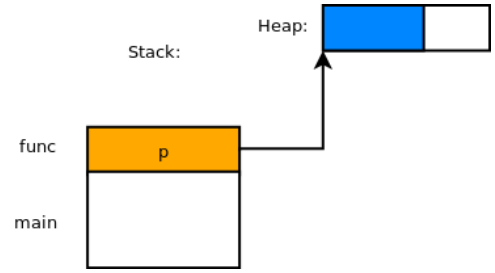
The interesting point of the example is that, in order to detect the Use-After-Free, the analysis needs to model the stack layout precisely. As we use the same region for all local variables from all functions (as explained in Section 4.6), it is straightforward to determine such relations. While classic static analyzers cannot detect this complex pattern, our model detects it directly, with Definition 8.

As they are hard to analyze, indirect Use-After-Free is not well known in the literature; yet modern exploits[Sil16] sometimes are based on this particular way of building Use-After-Free.

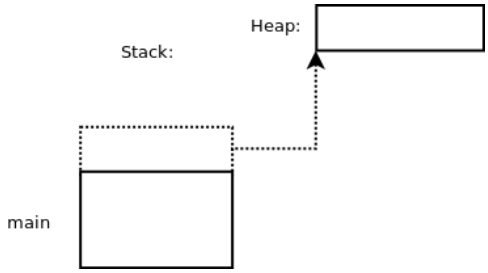
[1] Recall that we give our definitions with respect to the *x86* architecture

```c
void func()
{
        int *p;
        p=malloc(sizeof(int));
        free(p);
}

void vuln()
{
        int *p;
        int *q;
        q=malloc(sizeof(int));
        *q=0;
        *p=42;
        printf("q : %d\n",*q);
           // prints 42
}

void main()
{
        func();
        vuln();
}
```

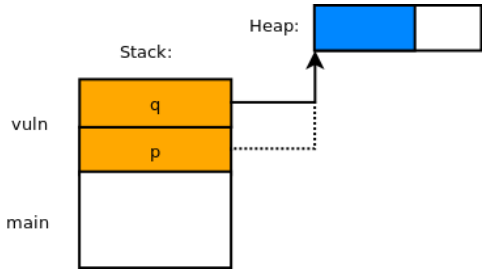
(a) Source code.



(b) In func(), after malloc()



(c) After func()



(d) In vuln(), after malloc()

Figure 5.5: Indirect Use-After-Free example.

5.4 Use-After-Free Representation and Grouping

From a CFG, Definition 8 (which defines $UafSet$, in Section 5.2) computes the set of Use-After-Free. However, when it is applied to real-world programs, this definition leads to numerous Use-After-Free detected (as we will demonstrate in the next Chapter), and it is not easily for a user to determine if these Use-After-Free are true positives. This section focuses then on the applicability of our analysis. Our goal here is twofold: first, we are interested in reducing the number of results to be analyzed, by grouping together results; secondly, we propose different visual representations of Use-After-Free, providing an *easy way* to understand the results for a user.

To this purpose, we detail in the following four techniques:

(i) The regrouping of *similar* Use-After-Free, leading to an extension of the Definition 8 (Section 5.4.1);

(ii) The slice extraction representing paths leading to Use-After-Free (in Section 5.4.2);

(iii) A method allowing a quick overview of the results, based on a tree representation (in Section 5.4.3);

(iv) Finally, this representation is used to create a signature, finding *likely to be similar* Use-After-Free (Section 5.4.4).

Benchmarks provided in Chapter 6 benefit from the representation detailed in this section; we focus here only on their descriptions and postpone the discussion on their efficiency for the next chapter.

In the following, the term analyst refers to the user of the analysis.

CFG operations To facilitate the definitions, we now introduce *root_alloc* and *root_free*, which return, for a node n and a *chunk*, the node where this chunk was allocated and the nodes where it was freed. While there exists a single node where a *chunk* is allocated, it can exist several nodes where it has been freed. The CFG operations are defined as follows:

$$root\_alloc: \ \mathbb{N} \times chunk \to \mathbb{N}$$
$$root\_freed: \mathbb{N} \times chunk \to P(\mathbb{N})$$

For example, in the motivating example in Figure 5.2 at line 35, for the first allocated chunk $chunk_0$, we have:

$$root\_alloc(35, chunk_0) = 11$$
$$root\_freed(35, chunk_0) = \{3\}$$

5.4.1 Regrouping Similar Use-After-Free

We propose here an extension of the Definition 8, merging *similar* Use-After-Free together.

Interest When dealing with binaries, the number of results obtained through static analysis can quickly grow up, due to the elevated number of false positives. Yet several results can correspond to the same part of the code.

For example, different use events can be detected for a single dangling pointer, thus leading to multiple similar results. In the motivating example in Figure 3.2, the set of Use-After-Free candidates contains two elements ($UafSet = \{(35, chunk_0), (36, chunk_0)\}$), yet for an analyst, they correspond to analyze nearly the same code. To improve the capacity of sorting the results, we propose to regroup results of the Definition 8 ($UafSet$).

Regrouping The idea here is to regroup results according to the similarity in their events (allocation / free / use). We propose three levels of grouping (formal definition are provided below):

- $UafSet_{use}$: two Use-After-Free are equivalent if the chunk used possesses the same allocation, free and use nodes;
- $UafSet_{free}$: two Use-After-Free are equivalent if the chunk used possesses the same allocation and free nodes;
- $UafSet_{alloc}$: two Use-After-Free are equivalent if the chunk used possesses the same allocation node.

These groups are expressed into the following equivalence relations:

$$(n_i, chunk_i) \sim_{use} (n_j, chunk_j) \quad \text{iff} \quad n_i = n_j \wedge chunk_i = chunk_j$$
$$(n_i, chunk_i) \sim_{free} (n_j, chunk_j) \quad \text{iff} \quad chunk_i = chunk_j \wedge root\_freed(n_i, chunk_i) = root\_freed(n_j, chunk_j)$$
$$(n_i, chunk_i) \sim_{alloc} (n_j, chunk_j) \quad \text{iff} \quad chunk_i = chunk_j$$

From these equivalence relations, we define an extension of $UafSet$ (from Definition 8), in Definition 10. $UafSet_g$, regrouping elements of $UafSet$ according to the equivalence relation $\sim_g$. Here N accounts for the set of *use* nodes that are regrouped.

Definition 10. Use-After-Free definition
$UafSet_g = \{(N, chunk) \mid \exists n_i, (n_i, chunk) \in UafSet \wedge N = \{n_j \mid (n_i, chunk) \sim_g (n_j, chunk),\ (n_j, chunk) \in UafSet\}\}$

Notice that Use-After-Free in $UafSet_{use}$ ($\sim_{use}$) correspond to $UafSet$ of Definition 8 ($UafSet_{use}$ contains one use node per element). $UafSet_{free}$ ($\sim_{free}$) allows gathering together Use-After-Free that differ only by the site of use, which corresponds to different usages of the same dangling pointer. $UafSet_{alloc}$ ($\sim_{alloc}$) aggregates together all Use-After-Free sharing the same allocation site. This definition is particularly useful to merge together different Use-After-Free that are false positives (see below).

Example From our motivating example (Figure 5.2), we have $UafSet = \{(35, chunk_0), (36, chunk_0)\}$, thereby:

- $UafSet_{use} = \{(\{35\}, chunk_0), (\{36\}, chunk_0)\}$;
- $UafSet_{free} = \{(\{35, 36\}, chunk_0)\}$;
- $UafSet_{alloc} = \{(\{35, 36\}, chunk_0)\}$.

Here, $UafSet_{free}$ and $UafSet_{alloc}$ are identical.

***$UafSet_{alloc}$* example** Let us consider the example in Listing 5.3. Here, two Use-After-Free are detected at lines 18 and 23 (both of which are false positives). Then, we have:

- $UafSet_{use} = \{(\{18\}, chunk_0), (\{23\}, chunk_0)\}$;
- $UafSet_{free} = \{(\{18\}, chunk_0), (\{23\}, chunk_0)\}$;
- $UafSet_{alloc} = \{(\{18, 23\}, chunk_0)\}$;

$UafSet_{free}$ does not allow to merge together these two Use-After-Free. Indeed, the free site of the Use-After-Free at line 18 is the *free* of the first call to `f1`, while the free sites of the second Use-After-Free contain the same free due to the first call to `f1`, but also a second free due to the second call to `f1`. It is thus not possible to merge these two cases by this representation. $UafSet_{alloc}$ on the opposite considers these two vulnerabilities as the same one. Then an analyst can discard this result by noticing that it corresponds to infeasible paths.

```
1  int f1(int *p)
2  {
3          if(cond)
4          {
5                  free(p);
6                  return -1;
7          }
8          return 0;
9  }
10
11 void f0()
12 {
13         int *p=malloc(sizeof(int));
14         if(1(p)==-1)
15         {
16                 return ;
17         }
18         *p=0;
19         if(1(p)==-1)
20         {
21                 return ;
22         }
23         *p=0;
24 }
```

Listing 5.3: $\sim_{alloc}$ example.

In the following, we refer as Use-After-Free, an element of $UafSet_g$ (Definition 10), represented by the couple $(N, chunk)$.

5.4.2 Slices Extraction

Interest To be usable, results of static analysis should be expressed to the analyst in a readable way. A possible solution, is by representing a given Use-After-Free (from $UafSet_g$) as a slice of the CFG containing all paths leading to the vulnerability. The slice will also be used to guide the DSE exploration, as discussed in Part III of this thesis.

Extraction Nodes of a slice can be divided into three types:

- (i) All nodes representing paths going from the entry node of the CFG to the allocation node;
- (ii) All nodes representing paths going from the allocation node to a free node;
- (iii) All nodes representing paths going from a free node to a node of use.

Therefore, we introduce the function $\mathcal{F}$, which returns all the nodes representing paths going from the set of nodes N_{src} to the set of nodes N_{dst}:

$$\mathcal{F}(N_{src}, N_{dst}) = \bigcup_{(src,dst)\in N_{src}\times N_{dst}} (succ^*(src) \cap pred^*(dst))$$

Then G' creates, from a Use-After-Free $(N, chunk)$, the corresponding slice:

$$\begin{aligned} G'(N, chunk) = (V', E'), with \quad v \in V' \quad & | \; v \in \mathcal{F}(\{entry\_node\}, \bigcup_{n\in N} root\_alloc(n, chunk)) && (i) \\ & \cup \mathcal{F}(\bigcup_{n\in N} root\_alloc(n, chunk), \bigcup_{n\in N} root\_free(n, chunk)), && (ii) \\ & \cup \mathcal{F}(\bigcup_{n\in N} root\_free(n, chunk), N), && (iii) \\ and \quad e_{i,j} \in E' \; | \; e_{i,j} \in E, i \in V', j \in V' & \end{aligned}$$

Example Recall that from our motivating example, we have:

- $UafSet_{use} = \{(\{35\}, chunk_0), (\{36\}, chunk_0)\}$
- $UafSet_{free} = \{(\{35, 36\}, chunk_0)\}$;

Figure 5.6a is the slice corresponding to the first element of $UafSet_{use}$ $(\{35\}, chunk_0)$. Figure 5.6b is the slice for its second element $(\{36\}, chunk_0)$. Figure 5.6c is the slice associated to the single element of $UafSet_{free}$ $(\{35, 36\}, chunk_0)$. The elements in dotted orange are the part which is removed. The three slices differ only in their last nodes (in particular in the *use* nodes, in red). As one can notice, the slice representation is a good illustration to understand the influence of the regrouping described in the previous section.

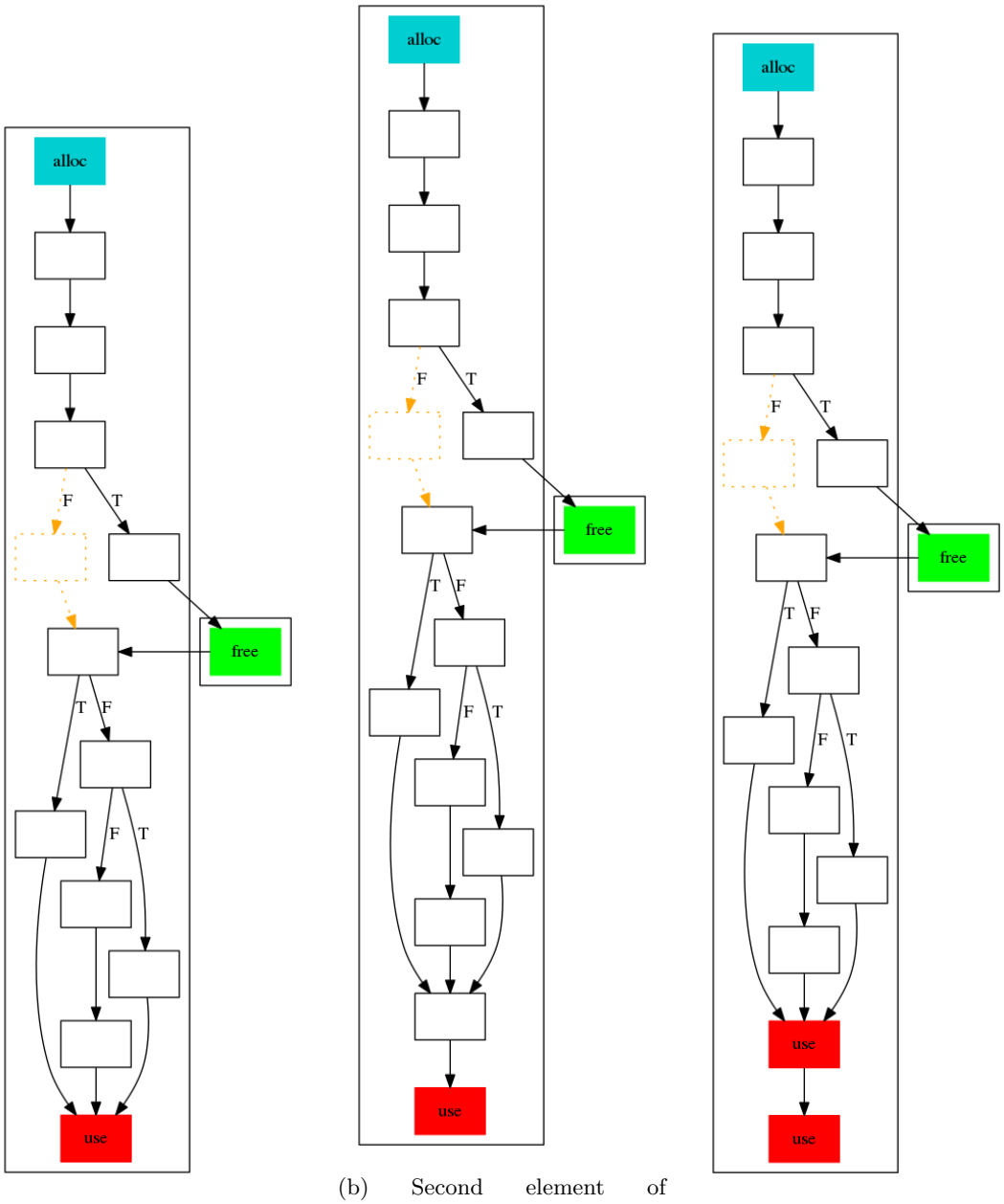
Loops As our model does not handle properly loop conditions, the exact number of loop iterations needed to trigger the Use-After-Free can not be determined by the VSA. Thereby we can not rely on the number of unrolling to validate if a slice represents a true positive. The last step of the slice extraction is then to rebuild original loop, by joining unrolled nodes corresponding to the same node. This is achieved by ignoring the iteration of nodes when exporting the slice.

Consider the example given in Listing 5.4, for which our analysis detects the Use-After-Free by unrolling the loop once.

```
1  for (..) {
2   if (cond1)
3    free(p);
4   else
5    ..
6  }
7  *p=42;
```

Listing 5.4: Loop example for slice extraction.

However, the precise number of iterations needed to trigger `cond1` is unknown; it is thus better to extract a slice containing the entire loop. Notice that DSE will determine the number of iterations needed to trigger the Use-After-Free at a later stage (as introduced in Section 3.1).



(a) First element of $UafSet_{use}$ (b) Second element of $UafSet_{use}$ (c) $UafSet_{free}$

Figure 5.6: Slice extraction for the motivating example.

5.4.3 Dynamic Calling Tree

We propose here a second visual representation of Use-After-Free, based on dynamic call tree.

Interest Slices can be used to understand the Use-After-Free at a *fine-grained view*, yet the slice can be large (in particular because of functions inlining). To overcome this problem, and to enable an analyst to picture the vulnerability easily, we propose a representation allowing to visualize it in a light way. Instead of showing all the nodes of the CFG, we are interested only in two types of nodes: (i) nodes corresponding to an event of the Use-After-Free (allocation / free / use), and (ii) nodes belonging to the call graph of the vulnerability. This representation gives then a *coarse-grained* view of Use-After-Free.

Dynamic Calling Tree We based our representation on the notion of dynamic calling tree [ABL97]. A dynamic calling tree (DCT) is an oriented rooted tree where each node represents a

call to a function. Our representation differs slightly from the classic definition of DCT: internal nodes represent a call to a function, while the leaves represent a specific event of the vulnerability (e.g.: an allocation, free or use node). Figure 5.7 represents the structure of the DCT. Events in the leaves are thus located in the function represented by their predecessor node.

$$
\begin{aligned}
node_{tree} &= call\_site \times son \\
son &= node_{tree} \mid event \\
event &= ins_{addr} \times it \times type \\
type &= Alloc \mid Free \mid Use
\end{aligned}
$$

Figure 5.7: Dynamic calling tree representation.

Example The DCT of the motivating example is given in Figure 5.8 for $\sim_{use}$ and in Figure 5.9 for $\sim_{free}$. This representation allows a quick overview of the vulnerability, as it focuses on how the functions are involved in the vulnerability.

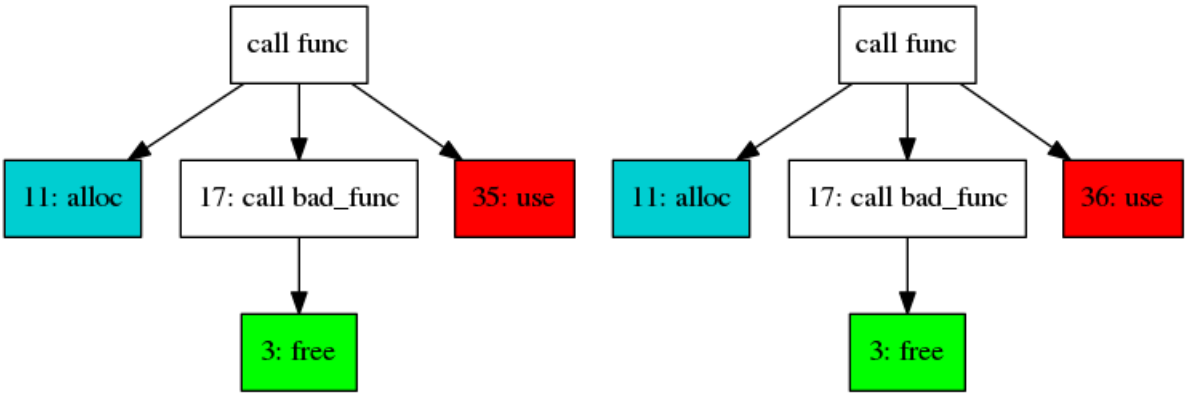


Figure 5.8: DCT of the motivating example using $\sim_{use}$, showing the allocation (blue), the free (green) and the use (red). $UafSet_{use}$ possesses two elements.

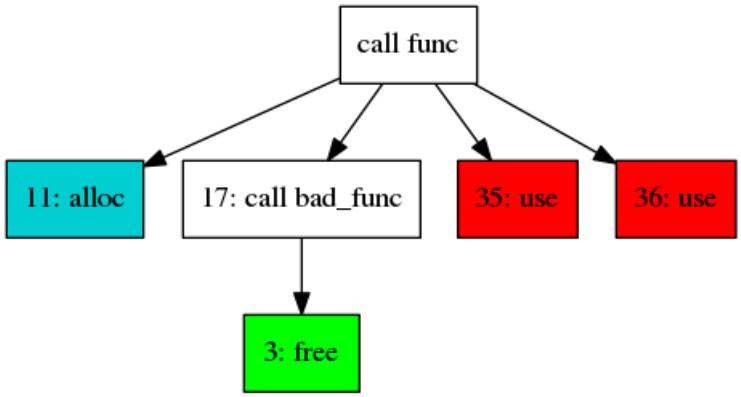


Figure 5.9: DCT of the motivating example, using $\sim_{free}$, showing the allocation (blue), the free (green) and the use (red). . $UafSet_{free}$ possesses one element.

5.4.4 Use-After-Free Signature

We discuss here the Use-After-Free signature that we developed.

Interest Two Use-After-Free (following Definition 10) can share the same part of the code, yet not be reached in the same manner. For example, as the analysis takes several entry points (as seen in Section 4.5.4), a function containing a Use-After-Free called by two different entry points would result in two different Use-After-Free. In such case, we cannot group Use-After-Free with equivalence relations defined in Section 5.4.1, are they come from different CFG. We propose then to compute a signature of Use-After-Free, to determine wich results share the same part of the code.

Signature This signature is based on a modification of the *DCT*: from it, we built a sub-tree with as root, the lowest common ancestor node of the allocation and free event nodes. The signature is used to compare Use-After-Free: two Use-After-Free with the same signature are *likely* to be similar (the term *likely* is explained below).

Example Consider a vulnerability inside a function `vuln`, which is called by two different entry point functions: `f0` and `f1`. Figure 5.10 shows the two *DCT* generated. As they come from two different CFG, it is not possible to regroup them. The signature from both vulnerabilities

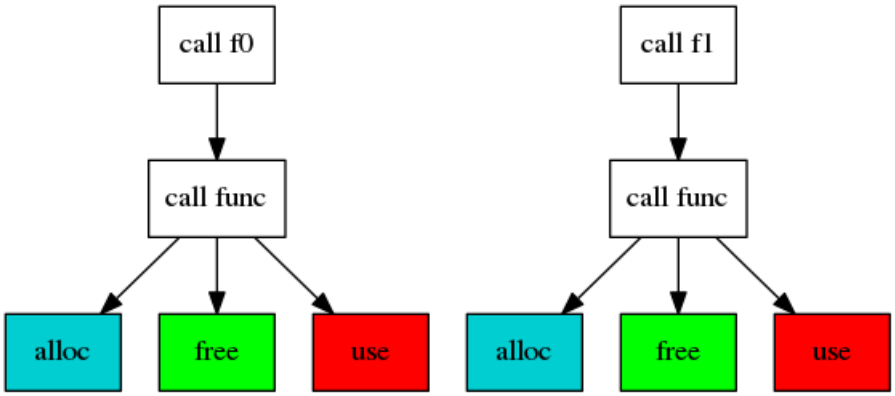


Figure 5.10: Same root cause, with two different *DCT*.

is represented in Figure 5.11, it is the same for both Use-After-Free. Thereby the analysis determines that these two results are *likely* to be similar.

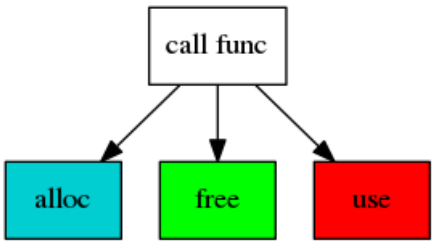


Figure 5.11: Signature of example Figure 5.10.

Limitation We use the term *likely* because, while two Use-After-Free can share the same signature, their calling context is different; the feasibility of their respective paths may be different. While in some cases analyzing only one Use-After-Free is sufficient to validate or invalidate Use-After-Free sharing the same signature, this is not always the case. Nevertheless, even if the results require to analyze each of the Use-After-Free, knowing which ones share the same code decreases the time needed by an analyst to evaluate the results.

5.5 Related Work on Heap State Modeling

This section discusses the related work on the heap model.

Heap model has been an active research area in these past years. Surveys on different techniques can be found in [Mar08; KK14]. A significant effort has been put into modeling the shape or the connectivity of heap objects, yet the purpose of such studies differs from ours. Some heap models focus on the lifetime of heap objects, more precisely seeking to find the last statement where a certain object can be addressed [Sha+03; GMF06; CR06]. However, these results differ from ours, as they look for the reachability of an object to determine when it can be safely freed; they do not track the status of the objects themselves. Static analyses detecting Use-After-Free mentioned in Section 2.3 track the status of the heap in a way which is generally not well detailed.

The allocation status model is, therefore, less studied and generally provided only through the source code of available tools. Nevertheless, we can mention [Wan16], which proposes a static analysis source code based on symbolic execution, tracking allocation status and detecting double free. Their model is close to the *object-based* representation and, as they rely on symbolic expressions to represent the program conditions, the example of Listing 5.1 does not lead to the same limitation as for our VSA. However, while static symbolic execution can be applied to source code analysis, it is not clear how such techniques would work with all the constraints of binary codes (such as the one mentioned in the previous chapter).

5.6 Conclusion

This chapter concludes the theoretical aspect of our static analysis. It presented the allocation status models developed and how they allow detecting Use-After-Free. More precisely, we have defined three allocation status models in Section 5.1.1, and Definitions 7 and 9:

- The *object-based* representation;
- The *pointer-based* representation;
- And finaly, a representation allowing the detection of stack-based Use-After-Free.

From these models, we have formalized how Use-After-Free are detected. Our models are precise enough to detect classic Use-After-Free as well as two different variants that are less studied by related work (stack-based and indirect Use-After-Free). We provided several heuristics reducing the number of results. Finally, we propose two representation helping to understand Use-After-Free detected: slices and *DCT*.

Chapter 6

GUEB: Implementation and Benchmarks

This chapter describes the implementation of the static analysis presented in Chapter 4 and Chapter 5, in a tool called GUEB (Graphs of Use-After-Free Extracted from Binary) and details several benchmarks obtained using this tool.

Outline

- First, Section 6.1 describes the design and architecture of GUEB;
- Section 6.2 shows the results of the tool on the examples provided in the previous chapters;
- Then, Section 6.3 provides results about new vulnerabilities found by GUEB in real-world programs;
- Section 6.5 addresses the question related to the performance of GUEB when applied at large scale;
- Finally, Section 6.6 concludes with a discussion on the limitations and perspectives for our static analysis.

All files necessary to reproduce the experiments of this chapter are available at `https://j-feist.com/thesis`.

In the benchmarks, we focus on the number of true positives (Use-After-Free detected and feasible) and false positives (Use-After-Free detected but not feasible) rather than on the number of false negatives (Use-After-Free not detected) or true negative (absence of Use-After-Free detected). Indeed, we lack a significant and available database of real Use-After-Free to evaluate our tool on these last two points. Moreover, we adopt the point of view that to be useful for security researchers, an analysis resulting in a low number of false positives is better than an analysis with a low number of false negatives.

6.1 Design

GUEB is the implementation of the static analysis described in the previous chapters. It is open source [Fei] and written in Ocaml. GUEB uses REIL [DP09] as an intermediate representation, which is provided through the BinNavi platform [Good]. This representation contains 17 instructions and has a syntax close to the *x86* assembly architecture. Currently, BinNavi can translate *x86*, PowerPc 32 bits and ARM 32 bits to REIL. BinNavi relies on IDA [Hex] as a disassembler. GUEB takes as input a Protobuf[1] file containing the exported REIL CFG of the program. This export is done through a graphical user interface written in Jython. Figure 6.1 illustrates the toolchain of GUEB.

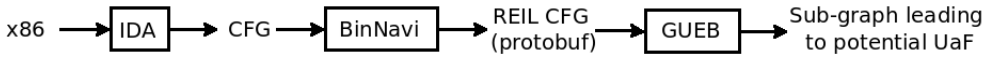


Figure 6.1: GUEB toolchain.

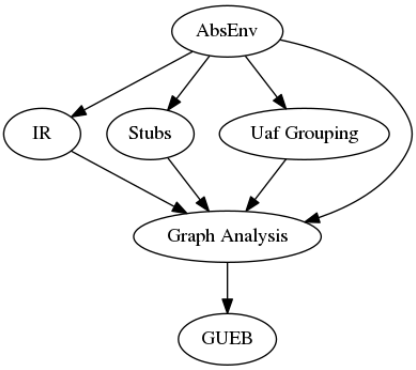


Figure 6.2: The modular architecture of GUEB.

Modular architecture The static analyzer is built using Ocaml functors, making it modular. Figure 6.2 shows the modules and their dependencies. The flexibility of the architecture in GUEB allows to easily prototype new analyses and to compare them. The *AbsEnv* module allows changing the memory model. GUEB possesses three memory models, accounting for the implementation of the different allocation status model described in Sections 5.1 and 5.3. The *Stubs* module allows creating specific stubs for analyses, for instance to model libraries or custom allocators. An example of its usage is detailed in Section 6.4. *Uaf Grouping* module specifies how to regroup the results, as detailed in Section 5.4.1 ($UafSet_{use}$, $UafSet_{free}$, and $UafSet_{alloc}$). *IR* module handles the intermediate representation and the transfer functions.

6.2 Validation of GUEB on Test Cases

We detail in this section the results of GUEB when applied to the different examples introduced in the previous chapters. This section demonstrates that GUEB works on the examples we used to illustrate the theoretical part of the analysis.

6.2.1 Motivating Example

We analyze with GUEB a compiled version of the motivating example presented in Figure 3.2a (Chapter 3), containing a function `main` calling `func`. The source code of this version is provided in Appendix A. As expected the Use-After-Free is detected.

Figure 6.3 is the *DCT* produced by GUEB (using $UafSet_{free}$). Each node contains the address of the instruction followed by its loop iteration number (always 0 in this example). For nodes representing a call, there is the called function name (or its address in case of stripped binaries). The blue leaf corresponds to the allocation instruction, the green is the free instruction and leaves in red are the different use locations of the dangling pointer.

Figure 6.4a represents the extracted slice. A slice contains one node per basic block; thus in this example, the two possible usages of the dangling pointer are represented through one basic block. Each external rectangle accounts for a function. GUEB provides another version of the slice, where edges between a `ret` instruction and the calling node are removed, as reflected in Figure 6.4b. While the first slice is needed for the DSE exploration, the second allows a better human understanding (the control-flow is generally more apparent). In the following, we use only the second representation, as we consider it easier to read.

As introduced in Section 5.4, *DCT* gives a *coarse-grained* view of the vulnerability, focusing on the location of the events accros functions, while the slice provides a *fine-grained* view.

Notice that addresses in the *DCT* and the slice are not the same; nodes of *DCT* contain addresses of instructions while nodes of the slice contain addresses of the basic block.

[1] https://github.com/google/protobuf

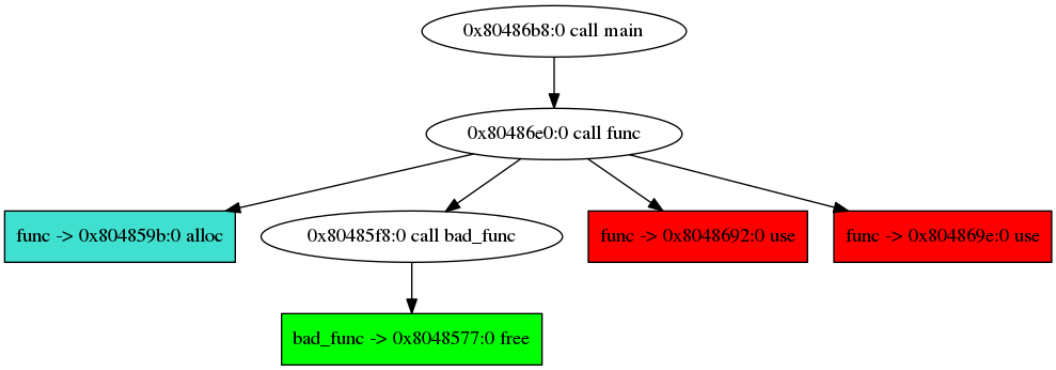
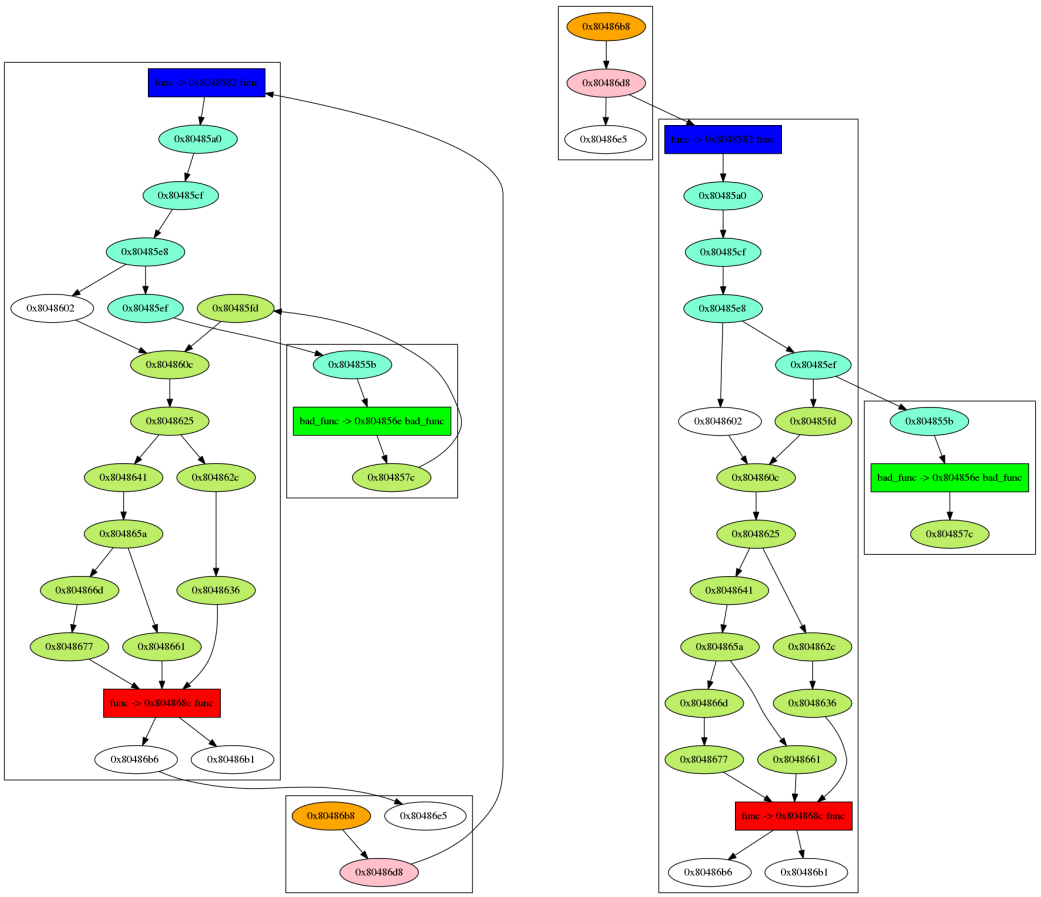


Figure 6.3: *DCT* of the motivating example.



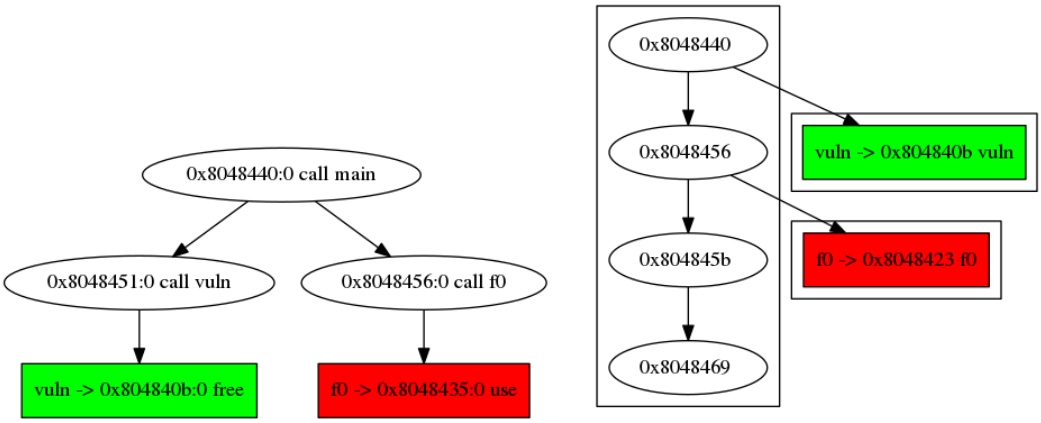
(a) With arcs from ret instructions.

(b) Without arcs from ret instructions.

Figure 6.4: Extracted slices of the motivating example.

6.2.2 Stack-Based Use-After-Free

The stack-based Use-After-Free presented in Section 5.3 (Figure 5.4) is detected by the memory model tracking this variant of Use-After-Free (Definition 9), while the other models do not detect it. Figure 6.5 shows the *DCT* and the slice produced by GUEB for this example. In the specific case of stack-based Use-After-Free, *DCT* and slices produced by GUEB do not possess any allocation site, as it corresponds to the prolog of the function in which the free occurs.

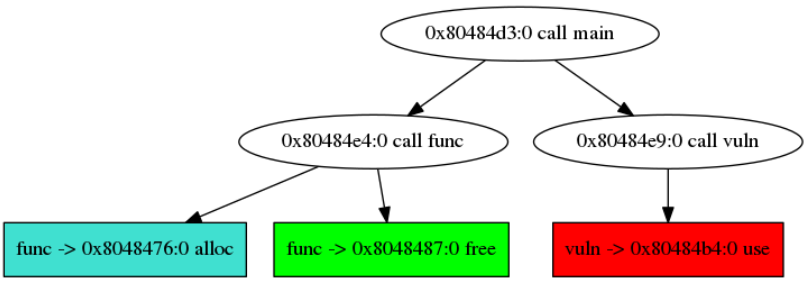


(a) *DCT*. (b) Slice.

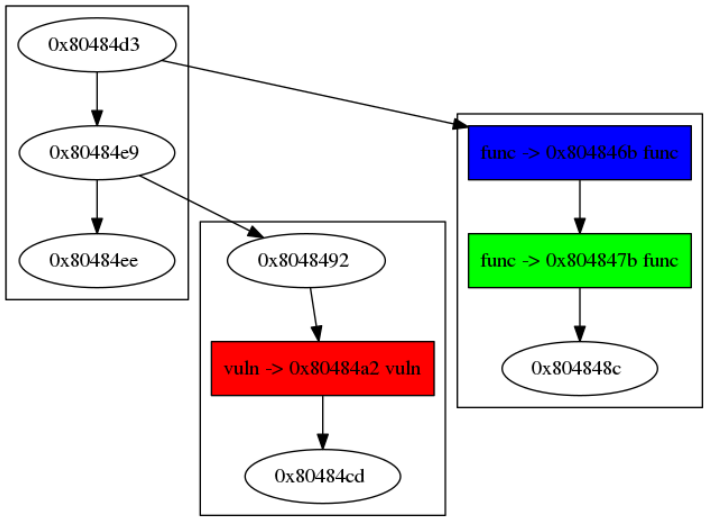
Figure 6.5: Results for the stack-based Use-After-Free example.

6.2.3 Indirect Use-After-Free

The indirect Use-After-Free introduced in Section 5.3 (Figure 5.5) is also detected by our tool. Figure 6.6 represents the *DCT* and the slice exported by GUEB in this case. As expected, the tool is powerful enough to detect such specific Use-After-Free pattern as it works at the binary level. As said before, tools based on source code analysis do not have a precise enough view of the stack layout and are not able to determine the effect of the use of uninitialized values.



(a) *DCT*.



(b) Slice.

Figure 6.6: Results for the indirect Use-After-Free example.

6.3 Finding New Vulnerabilities

There is no standard benchmark to evaluate the effectiveness of a static analysis for detecting Use-After-Free. As discussed in Section 3.5, classic vulnerability benchmarks generally contain only a small number of instances of this vulnerability. Thus there is no obvious mean to show the capacity of the analyzer to give an acceptable number of false positive while analyzing a large code. To demonstrate the efficiency of our tool for detecting new Use-After-Free, with an acceptable number of manual operations, we applied GUEB to several software to find new vulnerabilities. Although this is not a precise measure of the capability of the tool, it shows that our technique can be applied to real-world examples.

6.3.1 Finding Previously Unknown Use-After-Free

During its development, GUEB was applied to several binaries. This allowed us to find the following previously unknown vulnerabilities in different open-source projects:

- giflib (CVE-2016-3177): https://sourceforge.net/p/giflib/bugs/83/
- Jasper-JPEG-200 (CVE-2015-5221): http://www.openwall.com/lists/oss-security/2015/08/20/4
- openjpeg (CVE-2015-8871): https://github.com/uclouvain/openjpeg/issues/563
- gnome-nettool: https://bugzilla.gnome.org/show_bug.cgi?id=753184
- accel-ppp: http://accel-ppp.org/forum/viewtopic.php?f=18&t=581
- alsabat: https://bugzilla.redhat.com/show_bug.cgi?id=1378419

Table 6.1 summarizes the results of GUEB when applied to these programs, using a standard laptop (Intel i7-2670QM, 2.20GHz). All of them are compiled using `gcc` with the *-g* flag (debug-

name	#lines	time	#UAF	#Signature	#EP	#REIL ins	max size reached
alsabat	~ 2000	$7s$	1	1	10	99933	0
gnome-nettool (*-OO*)	~ 6500	$16s$	4	2	56	226514	0
gnome-nettool	~ 6500	$17s$	7	5	76	260882	0
gifcolor	~ 9000	$21s$	15	12	13	233303	0
jasper	~ 34200	$4m23$	255	114	205	2154927	5
accel-ppd	~ 61000	$5m5$	35	30	299	3907862	0
openjpeg	~ 205200	$6m10$	329	300	305	2170081	12

Table 6.1: Results of GUEB using $UafSet_{free}$.

ging information), except for gnome-nettool which was compiled in two versions: with *-g* and with *-g -OO* (without any optimization, this particular example is detailed below). The analysis was done with the *pointer-based* representation (see Section 5.1), and followed the $UafSet_{free}$ definition (see Section 5.4.1). We perform a function inlining based on the heuristic *bounded by size* (see Section 4.4), with a maximum number of instructions per entry points of 100,000. *#lines* is the number of lines of all the `C` files of these software[2]. *#EP* is the number of entry points of the analysis. *#REIL ins* is the number of REIL instructions analyzed in total (all entry points of the binary considered). *#Signature* is the number of distinct signatures of Use-After-Free (see Section 5.4.4). *max size reached* corresponds to the number of time the maximum size (100,000 instructions) is reached. As a comparison, Table 6.2 and 6.3 show the number of results found by GUEB when following the $UafSet_{alloc}$ and $UafSet_{use}$ definition. In particular, $UafSet_{use}$ results in a substantial number of results different to analyze; regrouping results is a clear advantage for an analyst.

[2]This number is obtained with a naive approach: all lines of the `C` files are counted, regardless if the line is empty or contains only comments.

name	#UAF	#Signature
alsabat	1	1
gnome-nettool (*-OO*)	4	2
gnome-nettool	5	3
gifcolor	15	11
jasper	114	39
accel-ppd	33	30
openjpeg	82	64

Table 6.2: Results of GUEB using $UafSet_{alloc}$.

name	#UAF	#Signature
alsabat	1	2
gnome-nettool (*-OO*)	5	3
gnome-nettool	5	9
gifcolor	786	10
jasper	6166	58
accel-ppd	481	30
openjpeg	4145	253

Table 6.3: Results of GUEB using $UafSet_{use}$.

Manual analysis of results To validate results of GUEB, a manual analysis by an analyst is needed. The fact that GUEB provides different representation and grouping of results helps the sorting of results. For example, for a given Use-After-Free, which an analyst found to be a false positive, he can conclude that it is interesting (or not), to analyze *similar* Use-After-Free and then focus (or not), to Use-After-Free sharing its signature. Such strategy can lead to find quickly a true positive, without having to actually analyze all the results.

Discussion Most of results here are false positives. Nevertheless, by manually analyzing them, we found true positives Use-After-Free in each of these programs. These results reveal that GUEB can be applied to real software with an acceptable number of false positives and with a low computation time. Even the largest example, `openjpeg` using $UafSet_{free}$, results in 300 different signatures, which is still acceptable for a manual analysis. While it is difficult to truly quantify the time spent to analyze each result, the fact that we actually found new vulnerabilities using GUEB demonstrates its effectiveness.

6.4 Case Study: gnome-nettool

This section details the study performed on gnome-nettool.

To facilitate the analysis, gnome-nettool is compiled using *-g, -OO* flags (thus keeping function names in the binary and removing optimizations). Removing optimizations helps the disassembler, which provides, thereby, a more accurate CFG. GUEB reports four Use-After-Free, with two different signatures (see Table 6.1). Three of these four Use-After-Free produce the same signature and are false positives. The fourth one is a true positive, and was reported to the developer of gnome-nettool. The DCT and the slice for this vulnerability are provided in Appendix B.

The second signature (leading to the three false positives) appears because of a missing call to a library. Listing 6.1 represents a simplification of the part of the source code leading to an imprecision in the results.

```
1  gchar *srtt_str;
2  while(cond){
3   gtk_tree_model_get (results, &node, SRTT_COLUMN, &srtt_str,
        -1);
4   ...
5   g_free (srtt_str);
```

```
6  }
```

Listing 6.1: get_bar_data from ping.c in gnome-nettool.

Here `gtk_tree_model_get` is a function belonging to the `GTK3` library[3]. As this function is linked dynamically, its code is not available in the binary code of gnome-nettool. Following the documentation of this library, it appears that a newly allocated object is put inside `srtt_str` at each loop iteration. This object is then freed by *g_free*. Yet, for GUEB *srtt_str* is never assigned and contains thus an uninitialized value, which is free twice (due to the loop unrolling), leading to detect a Use-After-Free. To avoid this false positive, a stub of `gtk_tree_model_get` needs to be considered. By using the stub given in the Appendix C, GUEB succeeds in removing this false positive; then, GUEB returns only one Use-After-Free for gnome-nettool: the true positive.

This example illustrates an important feature of GUEB: the customization of the analysis using stubs. Indeed, in practice, we observe that several false positives are detected due to missing calls or function leading to an erroneous computation. The usage of stubs helps the user to (iteratively) get rid of several false positives.

6.5 Scalability of GUEB

To show the efficiency of GUEB on real world software, and stress its performance, we have performed the analysis of a significant number of binaries. While it is possible to analyze the results of GUEB when applied a few examples, it is not feasible to verify the validity of so many binaries in a suitable amount of time. The goal of this section is to show that GUEB yields an *acceptable* number of results when it is applied to a binary. By *acceptable* we consider a number of results in the same order of magnitude that from the experiments provided in the previous sections. Thus, Use-After-Free detected in the following are more likely to be *false positives*.

Experimented setup We select binaries available in the directory `/usr/bin` (and sub-directory) of the version 16.04 of Ubuntu. We only retain binaries with calls to the function free of the *libc*[4]. 488 binaries have been converted to a protobuf file. For each of them, we perform four experiments:

- XP_1: With the *object-based* representation and the $UafSet_{alloc}$ definition;
- XP_2: With the *pointer-based* representation and the $UafSet_{alloc}$ definition;
- XP_3: With the *stack-based* Use-After-Free representation and the $UafSet_{alloc}$ definition;
- XP_4: With the *pointer-based* representation and the $UafSet_{free}$ definition.

All experiments used the inlining heuristics *bounded by size*, with a maximum of 200,000 REIL instructions per entry point, and a maximum of 1,000,000 REIL instructions per binary. The experiments were done on an Intel Xeon CPU E5-2650, with a limit of RAM to 5 GB.

Experimental results Table 6.4 shows the results of the experiments. 33 binaries are not

Experiment	# Bin	Without UAF	#UAF	#Signature	Time Total ($\bar{x}$,med,max)
XP_1	488	360	1379	712	1h 53m 31s (13s,3s,4m 43s)
XP_2	488	406	788	418	1h 49m 38s (13s,3s,4m 58s)
XP_3	455	351	1185	473	1h 31m 42s (12s,2s,2m 47s)
XP_4	488	406	1372	764	1h 49m 1s (13s,3s,4m 27s)

Table 6.4: Results of GUEB on 488 binaries from Ubuntu 16.04.

represented on the third experiment, as their analyses failed, due to an *out-of-memory* issue.

[3] https://developer.gnome.org/gtk3/stable/GtkTreeModel.html#gtk-tree-model-get
[4] Some binaries are also missing due to implementation errors, either inside BinNavi or our exporter.

The column *Without UAF* is the number of binaries where no Use-After-Free is reported. Here, $\bar{x}$ is the mean, while *med* the median.

Table 6.5 details the number of REIL instructions analyzed per experiments. The first two lines are the results on all binaries, while the last three lines are the results only on binaries for which a Use-After-Free has been found. This table shows that GUEB analyzed a large amount of REIL instructions. One can notice that, by removing binaries without Use-After-Free, the mean and the median grows up, meaning that, in average, binaries without Use-After-Free detected are smaller.

Xp	# Bin	Total	$\bar{x}$	med	max
$1, 2, 4$	488	51,160,172	104,836	17,501	998,630
3	455	43,212,311	94,972	14,769	998,630
1	128	33,455,291	261,369	231,476	998,630
$2, 4$	82	23,464,767	286,155	224,632	998,630
3	103	23,464,767	286,155	224,632	998,630

Table 6.5: Number of REIL instructions analyzed.

Comparison between experiments All experiments share the same order of magnitude for the computation time. The first experiment is the only one using the *object-based* representation and, as expected, it contains the greater number of Use-After-Free found. As a comparison, the second experiment differs only by using the *pointer-based* representation and results in almost detecting half of the results in comparison with the first experiment. The third experiment contains the same results as the second experiment[5], but also *stack-based* Use-After-Free. This experiment shows that the overhead in time and space brought by tracking *stack-based* Use-After-Free is low. Finally, a comparison between the second and the fourth experiment shows that there is an actual difference in term of the number of results between an experiment based on the $UafSet_{alloc}$ definition and on the $UafSet_{free}$ definition.

Repartition of results The repartition of Use-After-Free found is not the same across all binaries, and this is illustrated in Table 6.6. This table gives the details on the mean, median and the maximum number of signature and Use-After-Free found only on binaries for which Use-After-Free has been found. Binaries without Use-After-Free been removed, as they would decrease means and medians.

Xp	# Bin	Signature ($\bar{x}$,med,max)	UAF ($\bar{x}$,med,max)	Time Total ($\bar{x}$,med,max)
1	128	$(5, 3, 78)$	$(10, 3, 180)$	1h 14m 41s (35s,29s,4m 43s)
2	82	$(5, 2, 78)$	$(9, 4, 108)$	0h 53m 21s (39s,28s,4m 58s)
3	104	$(4, 2, 78)$	$(11, 5, 145)$	0h 50m 44s (29s,17s,2m 47s)
4	82	$(9, 3, 210)$	$(16, 5, 247)$	0h 52m 48s (38s,27s,4m 27s)

Table 6.6: Results of GUEB on binaries containing Use-After-Free.

Figure 6.7 illustrates the number of distinct signatures compared to the number of REIL instruction analyzed for the second experiment. It shows that the number of distinct signatures is not directly proportional to the number of instructions analyzed.

Discussion These benchmarks demonstrate that our method can be applied at large scale. In practice, Use-After-Free is a rare vulnerability; the fact that our tool does not find Use-After-Free in most of the binaries highlights its accuracy. While the tool does not guarantee the absence of vulnerabilities, the opposite (finding Use-After-Free in almost all binaries) would have denoted a bad result for our method. The number of instructions analyzed affects the number of results (more instructions to analyze means more chances to detect a Use-After-Free); yet as

[5]Except for binaries making the analysis failed due to an *out of memory*.

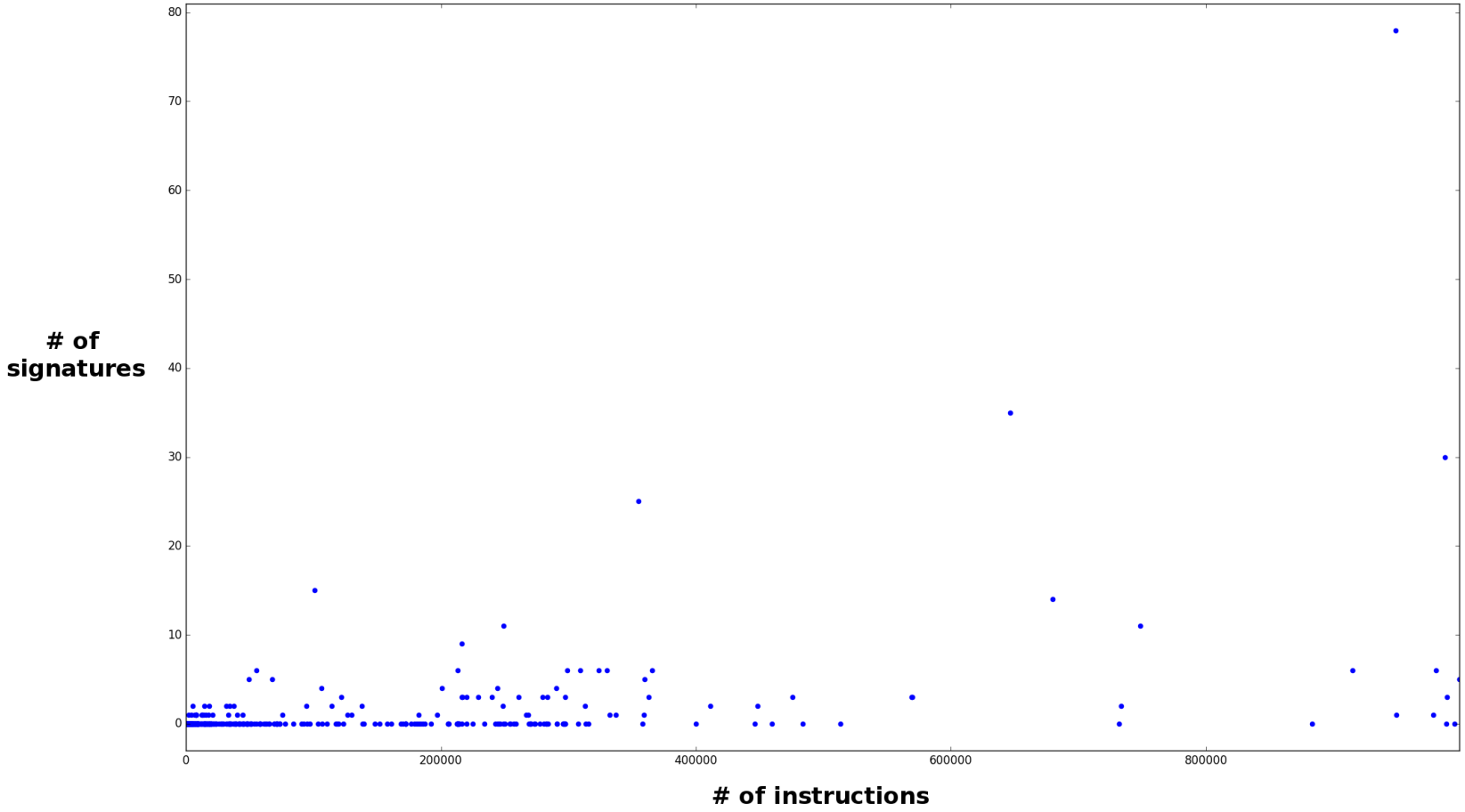


Figure 6.7: Number of distinct signatures per number total of instructions analyzed (experiment 2).

illustrated in Figure 6.7, the number of Use-After-Free found is not strictly dependent on the number of instructions analyzed.

We made the following observations:

- Our method is robust enough to detect true positive when applied at large scale with a low number of false positives;
- A *pointer-based* representation gives a better precision than an *object-based* representation (over 42% of results removed), without bringing an additional overhead;
- Tracking stack objects adds an acceptable space and time overhead;
- $UafSet_{alloc}$ gathers together around 60% more Use-After-Free representations than $UafSet_{free}$;
- The use of signature helps an analyst; finding the same Use-After-Free reachable in different manners is frequent, reporting them as equivalent reduces the number of results up to two or three times.

6.6 Limitations and Perspectives

We conclude this part with a discussion on current limitations of our analysis and possible perspectives.

Limitations Our analysis shows good performances for finding Use-After-Free while requiring a minimal amount of human intervention. However, it is not yet feasible to apply GUEB on very large binary codes, such as web servers or browsers; in these cases, the time and the number of false positives would be too large. Nevertheless, applied on some parts of the program, with an inlining heuristic bounded with a small depth or size (see Section 4.4) could be a solution to analyze large programs. Notice that GUEB can detect Use-After-Free in multi-threading applications if the Use-After-Free is not dependent of threads; otherwise, the static analysis developed is not adapted to track values between threads.

During its usage, we have also found that some types of software are not well suited for our value analysis. For example, programs based on *reference counters* results in significantly more false positives than other programs. The over-approximation is mainly due to the fact that we do not keep track of path conditions and do not take into account checks performed on the counter. Thus it leads to consider several infeasible paths.

Another limitation is that GUEB does not provide any coverage information; as the inlining is bounded, some part of the program could remain unanalyzed.

Finally, GUEB has some implementation restrictions, which could be easily overcome. For example, it supports only the *cdecl* calling convention, yet adding other calling conventions would not be difficult.

Perspectives The tool could be improved by adding a lightweight local path-sensitive analysis where needed. Indeed, a common source of false positives comes from examples similar to Listing 6.2. Here, the function `f` returns a specific value when it frees variables and does not free variables otherwise. The caller checks the returned value and never use the pointer if it is freed. However, our analysis considers a possible path from the free to a following use statement. Such a pattern is frequent and could be integrated into our analysis to avoid reporting such false positives. Providing a specific memory model, more precise and path sensitive, that would be applied only when such patterns appear, is a promising approach.

```
1  int f(int*p){
2   if(error){
3    free(p);
4    return -1;
5   }
6   else
7    return 0;
8  }
9
```

```
10  void caller(){
11   ..
12   if(f(p) == -1) {exit();}
13   // use of p
14  }
```

Listing 6.2: Example of false positive due to path sensitivity.

Other heuristics to deal with errors or approximations coming from the CFG recovery could also be incorporated. For example, authors of [MM16] describe an algorithm detecting all non-returning functions (such as functions always calling `exit`). Defining several other heuristics to tackle issues coming from the CFG of binary code is an interesting work direction.

Experiments were done using inlining following the *bounded by size* heuristic. Analyzing Use-After-Free detected by GUEB demonstrated that they are also detected using the *bounded by depth* heuristic with a low depth (less than four). It would be interesting to study which inlining strategy is better to detect Use-After-Free. However, such studies would require a larger database of real Use-After-Free.

GUEB has been developed with the intention of being coupled with a DSE engine, yet other usages are possible. For example, coupled with a mitigation system, it could locate parts of the program where possible Use-After-Free appear. Thereby, the mitigation system could choose to deploy costly memory protections only on a sub-part of the program, decreasing its overhead.

Finally, memory models developed are also particularly well adapted to detect the consequence of uninitialized values. Thus another interesting work direction could be adapting GUEB to focus on uninitialized values and their consequence instead of Use-After-Free.

Static Analysis: Summary of Contributions

This part has presented the theoretical and practical aspects of our static analysis. We recall here contributions made:

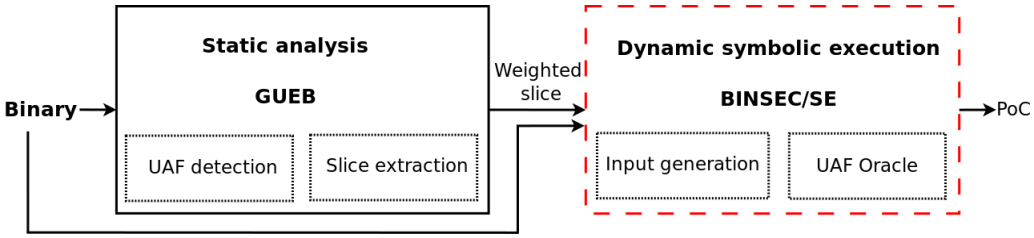
- The design of a memory model and a VSA well adapted to be applied to binary code for scalable analysis (Chapter 4);
- The study of the modelisation of heap objects and how different modelisations influence the detection of Use-After-Free, including variants of Use-After-Free (Chapter 5);
- Several representations of Use-After-Free to supply suitable results (Chapter 5);
- The open-source implementation of the analysis (Chapter 6);
- The discovery of new vulnerabilities in open-source projects (Chapter 6);
- The demonstration of the scalability and robustness of the implementation (Chapter 6).

Part III

Guided Dynamic Symbolic Execution

Guided Dynamic Symbolic Execution: Outline

The previous part described the static analyzer detecting Use-After-Free on binary code. While it demonstrates its efficiency to detect such vulnerabilities with a low number of false positive, it does not provide any possibility to show how to trigger the Use-After-Free automatically. The current part of this dissertation presents the guided dynamic symbolic execution developed to be applied directly on the results of GUEB. It is the second part of our global approach, as illustrated in red in the following figure:



The description of guided DSE is organized in three chapters. The first two chapters cover the theoretical aspects of the proposed approach, while the last one presents the implementation and the benchmarks performed.

Outline and contributions

- Our main contribution, an *end-to-end* approach combining the static analysis with guided DSE, is described in Chapter 7;
- While a general DSE allows to explore programs, it requires some adjustments to be applied on real examples. Chapter 8 details several refinements on DSE developed during this thesis;
- Finally, Chapter 9 focuses on the implementation of the guided DSE into the Binsec/SE platform and provides benchmarks obtained in combination with GUEB. It details the creation of a real *proof-of-concept* on a CVE obtained by GUEB, showing the effectiveness of our approach.

Chapter 7

Guided Dynamic Symbolic Execution

This chapter presents how to guide dynamic symbolic execution (DSE) in order to trigger Use-After-Free using the information produced by our static analysis.

Outline

- First, Section 7.1 recalls what is DSE and gives the definitions used in the remainder of this document;
- Section 7.2 defines the core of our contribution: how DSE is guided in order to trigger a Use-After-Free using a slice computed by the static analysis;
- Section 7.3 provides the Oracle validating the presence of a Use-After-Free in a trace;
- Section 7.4 discusses the validity of the results produced by our guided DSE;
- We discuss related work in Section 7.5;
- Finally, Section 7.6 concludes this chapter with a discussion of our contributions.

7.1 Dynamic Symbolic Execution: Background

Dynamic Symbolic Execution, also called concolic execution, is a program analysis technique which consists in representing the program traces as a set of constraints over the program inputs [GKS05; SMA05; GLM12; Cha+12]. It is inspired by the symbolic execution technique proposed in [Kin76], where program paths are expressed by path predicates, i.e., logical formulas over the program variables. The specificity of DSE is to mix within the path predicates some concrete information coming from a concrete execution trace. A common purpose of DSE is the automatic exploration of the program; by inverting conditional branches on path predicates it can generate inputs allowing to execute new traces.

The main features of DSE exploration are:

- The path predicate computation: using concretization strategies [God11; Dav+16b], it is possible to decide which data should be considered as **symbolic** or **concrete** when building a path predicate.
- The trace exploration strategy: a complete exploration of all traces being not always realistic, several heuristics have been designed to address this limitation [GLM12; Ma+11].

The path predicate computation involves several features. In this chapter, we only give an overview of the path predicate generation. Various improvements on the path predicate generation are detailed in the next chapter.

Section 7.1.1 first describes what a path predicate is. Then a discussion on how to solve the path predicates is provided in Section 7.1.2. Finally Sections 7.1.3 to 7.1.5 describe the DSE exploration.

7.1.1 Path Predicate

Definitions We recall here the Definition 5 of Chapter 4. A node n in the program CFG is represented as a triplet $ins_{addr} * it * cs$, where ins_{addr} is the address of the instruction, it, the number of iteration after loop unrolling, and cs the call stack.

An execution trace t is a sequence of nodes n:

$$t = n_0 ... n_i ... n_n$$

A path predicate Φ_t is the logical conjunction of all constraints induced by the effects of nodes statements along the trace t. We can distinguish between two kinds of statements: assignments and branching conditions. On the constraints, program inputs are expressed by existentially quantified symbolic variables. Input functions determine which memory locations are expressed by such variables. In the following, we use the static single assignment form to express variables.

Let Φ_{t_i} be the path predicate built from n_0 to n_i of the trace t.

Example of path predicate Let us illustrate the path predicate computation on the example of Listing 7.1.

```
1  // x,y are inputs
2  int z;
3  z = x+1;
4  if(z>y)
5    z=0;
6  else
7    z=1;
```

Listing 7.1: Example of path predicate computation.

Consider the execution trace t[1], generated from inputs $x = 2; y = 0$:

$$t = n_3, n_4, n_5$$

From this trace, we can compute (in bold is the effect of the last node to the path predicate):

$$\Phi_{t_3} \triangleq (\mathbf{z_0 = x_0 + 1})$$
$$\Phi_{t_4} \triangleq (z_0 = x_0 + 1 \wedge \mathbf{z_0 > y_0})$$
$$\Phi_{t_5} \triangleq (z_0 = x_0 + 1 \wedge z_0 > y_0 \wedge \mathbf{z_1 = 0})$$

Symbolization Program inputs are expressed through symbolic variables (here x_0 and y_0), yet during the path predicate computation any memory location can be assign to a fresh symbolic variable; this process is called *symbolization*. Symbolization of non-input memory locations have an interest for some use of symbolic execution (such as in the presence of uninterpreted functions [God11]). For our purpose of program exploration, we only symbolize program inputs and the initial memory.

Concretization As DSE is based on concrete execution traces, path predicates can be updated with concrete values coming from the trace; this process is called *concretization*. For example, while y is an input in the previous example, we can concretize its value with the value of the initial concrete execution trace t (here, $y = 0$). Paths predicates then become:

$$\Phi_{t_3} \triangleq (\mathbf{y_0 = 0} \wedge z_0 = x_0 + 1)$$
$$\Phi_{t_4} \triangleq (\mathbf{y_0 = 0} \wedge z_0 = x_0 + 1 \wedge z_0 > y_0)$$
$$\Phi_{t_5} \triangleq (\mathbf{y_0 = 0} \wedge z_0 = x_0 + 1 \wedge z_0 > y_0 \wedge z_1 = 0)$$

Concretization has multiple applications. It can be used to reduce the path predicate size (for example, by concretizing the return value of a function, it can allow to skip its execution) or to develop specific uses of DSE (such as the Oracle presented in Section 7.3).

[1] Here, the index of a node corresponds to its line in the source code.

7.1.2 Solvers

A popular solution to solve path predicates is to use SMT (satisfiability modulo theories) solvers. SMT solving is still an active research area [Oli14; Mon16]. In our implementation, we use the SMTLib standard format[2], which allows using different solvers against the same formula (such as Z3 [MB08], boolector [NPB15a], yices [DD06]).

Solver output If the path predicate is satisfiable (SAT), the solver generates an instantiation of free variables satisfying the path predicate, called a *model*. If it is not, the solver returns $UNSAT$ (for unsatisfiable). Solvers are generally bounded in time; if a solver does not conclude with the satisfiability of the path predicate after a finite time, it generates a *timeout*.

Correctness and completeness of a path predicate Definition 11 formalizes the correctness and the completeness of a path predicate. In particular, concretization and symbolization operations presented in the previous section have impacts on these properties. More details can be found in [God11; Dav+16b]. Due to our goal (generating inputs to explore the program), the correctness of path predicates is crucial, while its completeness is not mandatory.

Definition 11. A path predicate Φ_t is said to be *correct* if all of its models give a program input valuation allowing to cover t. Conversely, Φ_t is said to be *complete* if each input valuation covering t is a model of Φ_t.

Controllability of inputs On models generated by a solver, we will distinguish two types of variables:

- Direct user-controlled variables, assigned by input functions;
- Variables associated to uninitialized memory locations.

Variables associated to uninitialized memory locations are not controllable by the users; thereby, to have correct path predicates (i.e., such a model allows to play the corresponding execution trace), DSE needs models containing only valuations on user-controlled variables. This problem is addressed in the next chapter (Section 8.4); we consider in this chapter formulas without uninitialized variables. *generate_input* is the function generating an input according to a model:

$$generate\_input : model \rightarrow input$$

SMTLib limitations There are several theories available to express path predicates in the SMLTLib standard; in the implementation presented in Chapter 9, our path predicates are built using the *bitvector* and *array* theories. The choice of using these theories induces some limitations. For example, due to the impossibility to combine forall ($\forall$) quantifiers with *bitvector*, some of our analyses are limited (see Sections 7.3 and 8.4). Extensions of the standard exist [WHM13; Dut15], supporting such quantifiers along with *bitvector*, yet they are solver-dependent, and we do not consider these possibilities in the following.

7.1.3 Trace Exploration Based on DSE

Trace exploration is achieved by inverting conditional branches on path predicates, leading to the generation of a new path predicate representing a potential new trace. Solving this path predicate may either produce some input valuation allowing to explore the corresponding trace, or lead to conclude that this trace is not feasible.

In the following, Φ'_{t_i} is Φ_{t_i} where the last *branching condition* has been inverted.

Example of conditional branch inversion Recall the example in Listing 7.1 from Section 7.1.1, with a trace t generated using $x = 2; y = 0$ ($t = n_3, n_4, n_5$). In this case, one node corresponds to a branching condition: n_4. Recall Φ_{t_4} from Section 7.1.1:

$$\Phi_{t_4} \triangleq (z_0 = x_0 + 1 \wedge z_0 > y_0)$$

[2] http://smtlib.cs.uiowa.edu/

If we invert the last constraint of Φ_{t_4} we obtain (in red the inversion):

$$\Phi'_{t_4} \triangleq (z_0 = x_0 + 1 \wedge z_0 \leq y_0)$$

This formula can be solved by a solver, which generates then a model satisfying the formula, for example:

$$x_0 = 0; y_0 = 10;$$

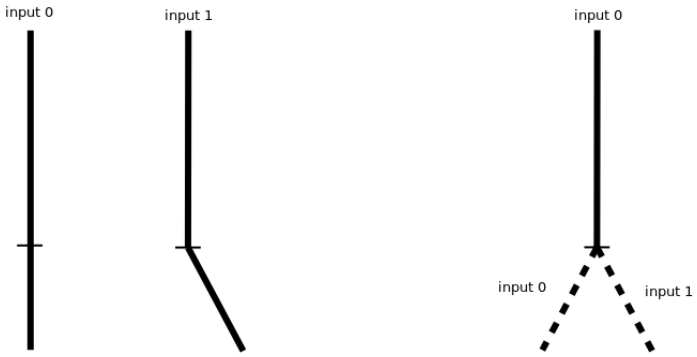
Then this model can be used to generate a new input, leading to explore a new trace.

7.1.4 Offline and Online DSE

DSE exploration can be classified into two types: *online* and *offline* (using the definition from [Cha+12]). We detail in the following the difference between these two types.

Offline DSE

In *offline DSE*, the program is executed with an initial concrete input, and the corresponding trace is recorded. The path predicate is then computed from this trace and new inputs may be generated from the inversion of branching conditions on this path predicate. Then these inputs permit the execution of new traces, which create new path predicates and so on, until a chosen termination condition is met. Figure 7.1a illustrates an offline DSE. Here, from a first input we generate a trace, containing one condition. The path predicate computed from this trace allows inverting the condition, which generates a new input, leading to a new trace.



(a) Offline DSE. (b) Online DSE.

Figure 7.1: DSE type illustration.

Online DSE

The second type of DSE is called *online DSE*. Here, starting also from an initial concrete input, the program is executed and the path predicate is built all along the execution. When a conditional instruction is met, the DSE forks the execution, updates the input using the corresponding path predicate and explore the two possible traces. Figure 7.1b illustrates an online DSE.

Offline versus Online DSE

Offline DSE suffers from the fact that all the instructions are executed for every input, while online DSE executes them only once; offline DSE is then slower than its online counterpart. However, online DSE brings its own problems, in particular, due to the forking operations. First, this operation is expensive in memory consumption. Moreover, forking makes keeping the consistencies of the environment harder (such as file systems or network sockets). For example, if a socket is shared by two paths, and closed in the first one, any reads from in the second path will lead to an error.

As we want to produce real inputs triggering vulnerabilities, the correct handling of the environment is crucial for us. Thus we decide to implement an offline DSE, to which we refer simply as DSE. All our algorithms are defined for offline DSE.

7.1.5 Traces Exploration Algorithm

A program P is viewed as a function producing an execution trace from an input:

$$P : input \rightarrow trace.$$

Trace exploration is achieved by iterating on program traces. A generic algorithm is provided in Algorithm 15. Here WL is the working list containing the couples (t, n), where t is an execution trace, and n a conditional node of this trace.

In a high-level perspective, the algorithm works as follow:

1. The program is executed from an initial program input i_0 (called a *seed*) to produce a first trace t_{i_0}; conditional nodes of t_{i_0} initialize the working list WL;
2. A couple (t, n), where t is a trace and n a conditional node, is selected and removed from WL;
3. The path predicate Φ'_{t_n} is computed by inverting the last condition of Φ_{t_n};
4. Φ'_t is then solved using an SMT solver: if the predicate is satisfiable, a solution is returned as a model m. This model generates then a new input i;
5. The input i allows to explore a new trace t_i; *some* conditional nodes of t_i are extracted and added to WL; then the algorithm resumes at step 2.

Algorithm 15: Generic trace exploration algorithm

Input: Program P, input seed i_0
$t_{i_0} := P(i_0)$;
Conditional nodes of t_{i_0} extracted and added in WL;
while $WL \neq \emptyset$ *and continue*() **do**
  select $(t, n_n) \in WL$;
  $WL := WL \backslash \{(t, n_n)\}$;
  $\Phi'_{t_n} := compute\_predicate(t, n_n)$;
  $verdict, m := ask\_solver(\Phi'_{t_n})$;
  if $verdict = SAT$ **then**
    $i = generate\_input(m)$;
    $t_i := P(i)$;
    Some conditional nodes of t_i are extracted and added in WL;
  end
end

The exploration terminates either when:

- WL is empty;
- A specific criterion is satisfied in a trace explored (e.g., a trace containing a vulnerability is found);
- A bound over the exploration is reached (e.g., a maximum number of traces to explore or a timeout).

As exploring all traces is not realistic for most programs [CS13], the exploration is generally bounded. This limitation is known as the *path explosion* problem. Step (2) (*select*) and the selection of conditional nodes to invert are then crucial, as they determine in which order traces are explored.

7.2 WS-Guided DSE

We describe in this section how to guide a DSE using the slice extracted from GUEB. This section is divided in two parts:

- GUEB produces slices that may contain loops (see Section 5.4.2), yet traces contain unrolled loops. As targeted events (allocation / free / use) can be located in a loop, we need to take into consideration some *projections* of the nodes of the traces which contains unrolled loops to the nodes of the slice which contains loops. We detail the handling of loops in Sections 7.2.1 to 7.2.3.

- To minimize the time spent to trigger Use-After-Free, the objective of our guiding technique is to drive DSE towards execution traces belonging to the slice and not explored yet. Our approach first produces a *weighted slice* (WS) from the original slice, using a metric computed as a preprocessing step of the exploration. Sections 7.2.4 and 7.2.5 detail the preprocessing and the exploration step.

7.2.1 Loops in the Slice

The following definitions determine how to project nodes of a trace to nodes of the slice. This projection is achieved by ignoring the loop iteration of the trace nodes. We define $=^{\downarrow}$ and $\in^{\downarrow}$ as the respective projections of the relations $=$ and $\in$ on $addr_{ins}$ and cs of nodes:

$$\begin{aligned} &=^{\downarrow} node \times node \quad \rightarrow bool \\ &\in^{\downarrow} node \times P\,(node) \rightarrow bool, \end{aligned}$$

where:

$$\begin{aligned} (addr_i, it_i, cs_i) =^{\downarrow} (addr_j, it_j, cs_j) &\triangleq addr_i = addr_j \wedge cs_i = cs_j \\ n_i \in^{\downarrow} N &\triangleq \exists n_j \in N,\ n_i =^{\downarrow} n_j. \end{aligned}$$

7.2.2 Node Definitions

Recall that a Use-After-Free is represented by the couple $(N, chunk)$ where N accounts for the set of nodes in the CFG where $chunk$ is used, as established in Definition 10 of Chapter 5.

Recall functions $root\_alloc$ and $root\_free$ from Chapter 5, which return, for a node n and a $chunk$, the node where this chunk was allocated and the nodes where it was freed.

In the following we use, for a given Use-After-Free:

- The allocations nodes, defined by $N_{alloc} = \underset{n \in N}{\cup} root\_alloc(n, chunk)$;

- The free nodes, defined by $N_{free} = \underset{n \in N}{\cup} root\_free(n, chunk)$;

- The use nodes, defined by $N_{use} = N$.

Our goal is to find an input triggering a trace t_{uaf} such that:

$t_{uaf} = n_0, .., n_{alloc}, ..n_{free}, ..., n_{use}, ..$ with $n_{alloc} \in^{\downarrow} N_{alloc}$, $n_{free} \in^{\downarrow} N_{free}$ and $n_{use} \in^{\downarrow} N_{use}$.

Notice that N_{alloc} is always a singleton.

7.2.3 Extracting Unexplored Edges of a Trace in the Slice

Given a trace, we want to explore its unexplored continuations; these correspond to the unexplored outgoing edges of the projection of the trace to the slice. This notion implies that:

(i) Some edges are not to be explored (e.g., edges leading outside the slice);

(ii) Events of the Use-After-Free (allocation, free and use) can appear in a loop; we need to differentiate two nodes of a trace located in the same loop depending if a Use-After-Free event has already occurred or not in the loop.

Partitioning First, we partition the trace t in four parts, with respect to Use-After-Free events (formal definitions are provided below):

- P_{use} = the set of nodes occurring between the first free node and the last use node;
- P_{free} = the set of nodes occurring between the allocation node and the first free node (without nodes in P_{use});
- P_{alloc} = the set of nodes occurring between the initial node and the allocation node (without nodes in P_{free} and P_{use});
- And finally, P_{out}, all the others nodes.

Let $first\_free$ be the function returning, from a trace, its first free node and $last\_use$ the function returning its last use node:

$$\begin{aligned} first\_free(t) &= n_i \mid n_i \in t, n_i \in^{\downarrow} N_{free} \wedge \forall j, n_j \in t, j < i \Rightarrow n_j \notin^{\downarrow} N_{free} \\ last\_use(t) &= n_i \mid n_i \in t, n_i \in^{\downarrow} N_{use} \wedge \forall j, n_j \in t, j > i \Rightarrow n_j \notin^{\downarrow} N_{use} \end{aligned}$$

We define the function *partition*, which splits a trace as follows: $partition(t) = \{P_{alloc}, P_{free}, P_{use}, P_{out}\}$ where

$$\begin{aligned} P_{use} &: \{n_i \in t \mid \exists j, k,\ j \leq i < k, n_j = first\_free(t), n_k = last\_use(t)\} \\ P_{free} &: \{n_i \in t \mid \exists j, k,\ j \leq i < k, n_j \in^{\downarrow} N_{alloc}, n_k = first\_free(t)\} \\ P_{alloc} &: \{n_i \in t \mid \exists k,\ i < k, n_k \in^{\downarrow} N_{alloc}\} \\ P_{out} &: t \setminus P_{alloc} \setminus P_{free} \setminus P_{use} \end{aligned}$$

If there is no free node in the trace, P_{use} is empty, and $first\_free$ returns the last node of the trace in the slice. Similarly, if there is no use node, but there are some free nodes, $last\_use$ returns the last node of the trace in the slice.

Note that *partition* produces an unique partition of a trace t. Several partitions are possible; here we choose to partition over the first free node and the last use node.

Example of partitioning Let us consider the slice in Figure 7.2, where all events of the Use-After-Free appear inside a loop and the trace t extracted from this slice, represented in Figure 7.3a. This trace contains the allocation node. The partition of t is illustrated in Figure 7.3b, where nodes in the blue rectangle are in P_{alloc}, while nodes in the green rectangle are in P_{free}. Here, P_{use} and P_{out} are empty.

Target Use-After-Free needs to reach the allocation, the free and the use events in sequence. As a result, when exploring a given trace, our guiding technique is driven by the *next Use-After-Free event to reach*. Thus we associate to each trace node a so-called *target*, as follows:

$$target: alloc \mid free \mid use \mid out$$

The function $get\_target\_to\_reach$ returns the target of a node n in the trace t according to the part it belongs to:

$$get\_target\_to\_reach: trace \times node \rightarrow target,$$

where:

$$get\_target\_to\_reach(t, n) = \begin{cases} alloc & if \quad n \in P_{alloc} \\ free & if \quad n \in P_{free} \\ use & if \quad n \in P_{use} \\ out & if \quad n \in P_{out} \end{cases} \quad with\ \{P_{alloc}, P_{free}, P_{use}, P_{out}\} = partition(t)$$

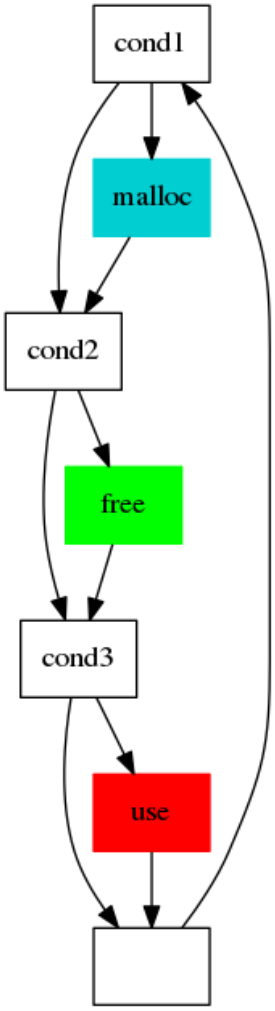


Figure 7.2: Slice example.

Unexplored edges We can now define *get_unexplored* as the partial function returning all unexplored edges of the projection of the trace t on the slice. Each unexplored edge is represented by a triplet $(n_{src}, n_{dst}, target)$, where (n_{src}, n_{dst}) accounts for the unexplored edges and $target$ is computed using $get\_target\_to\_reach(t, n_{src})$. Its prototype is:

$$get\_unexplored : trace \rightarrow P\left(node \times node \times target\right),$$

where

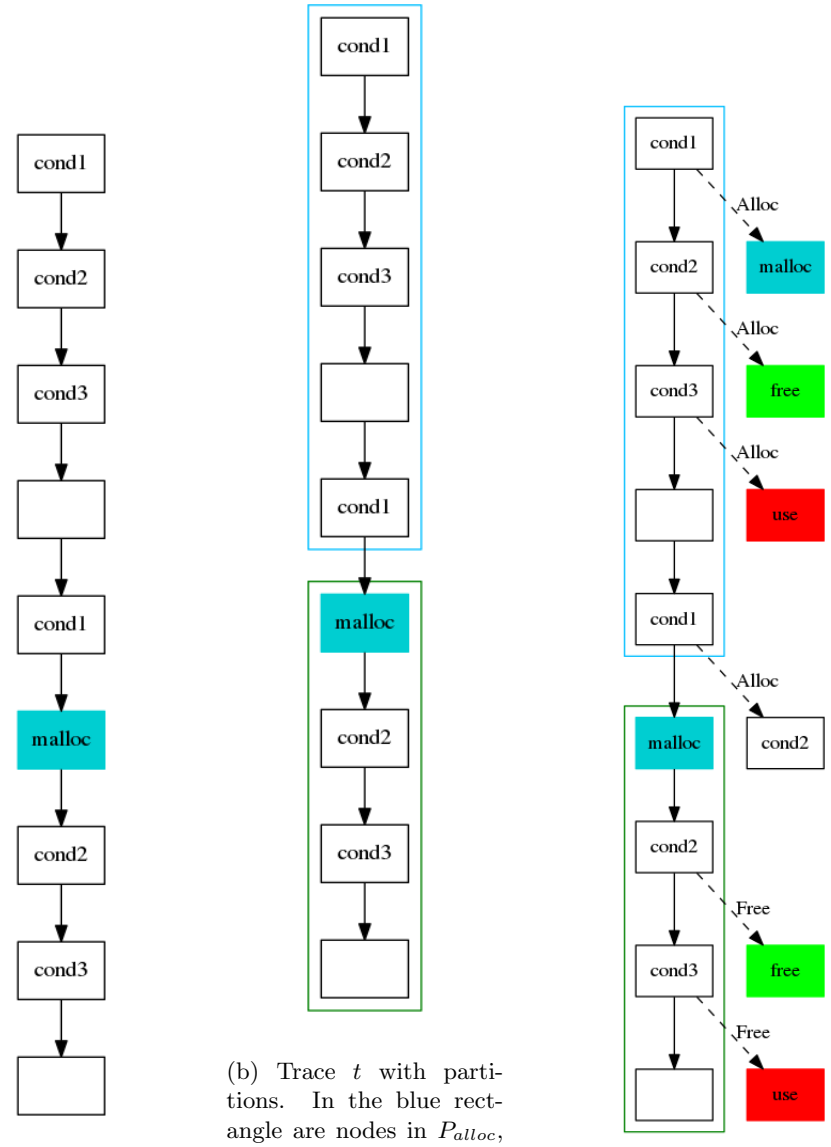
$$get\_unexplored\_(t) = (n_i, n_j, t_g) \quad | \; t_g = get\_target\_to\_reach(t, n_i), \\ n_i \in t \wedge n_j \notin t \wedge (e_{i,j} \in E_s) \wedge t_g \neq out.$$

E_s is the set of edges of the slice to explore. *get_unexplored* does not return any triplet if its target is *out*.

Unexplored edges example Figure 7.3c illustrates the unexplored edges on the previous example. Here four unexplored edges lead to the allocation node and two to the free nodes.

7.2.4 Weighed Slice

To prioritize between unexplored edges, we compute a metric as a preprocessing step before the exploration.



(a) Trace t.

(b) Trace t with partitions. In the blue rectangle are nodes in P_{alloc}, in the green rectangle are nodes in P_{free}

(c) Trace t with unexplored edges.

Figure 7.3: Unexplored edges extraction example.

Distance Score We denote the score to the targets as DS (*Distance Score*). As the trace is partitioned in three parts[3], we compute three scores for each target (*alloc*, $free$ and use). The prototype of DS is:

$$\begin{aligned} &score\colon \mathbb{N} \\ &DS \quad : (node \times node) \times target \to score. \end{aligned}$$

The first argument of DS is an edge (represented by its node source and its node destination), and the second is the targeted node. Algorithm 16 defines DS. Here, $Weight_{target}$ is the metric computed as preprocessing. In our implementation, we based the metric on shortest paths in the slice or random walks (as described in Section 8.1); yet our approach is generic and other metrics could be used.

Example of score Figure 7.4 illustrates the score computed from the example introduced in Figures 7.2 and 7.3; here, the metric is the shortest path based on the number of edges to the

[3]It is in four, but we do not consider the P_{out} element here.

Algorithm 16: *DS*

```
DS((src,dst),target)
    switch target do
        case alloc do
            return Weight_alloc(src, dst)
        end
        case free do
            return Weight_free(src, dst)
        end
        case use do
            return Weight_use(src, dst)
        end
    end
    return undefined
```

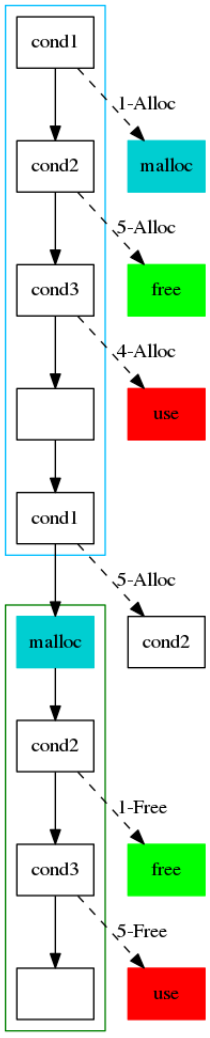


Figure 7.4: Score example

destination. In this example, the unexplored edges ($cond2$,$free$) do not have the same score depending if the allocation already occurred or not:

- If the allocation did not occur, $cond2$ is in P_{alloc}, and the score to the target ($alloc$) of the unexplored edge ($cond2$,$free$) is 5;

- If the allocation did occur, $cond2$ is in P_{free}, and the score to the target ($free$) of the unexplored edge ($cond2$,$free$) is 1.

7.2.5 Traces Exploration Algorithm

Algorithm 17: WS-Guided DSE

Input: Program P, DS, oracle σ, input seed i_0
Output: Input i, with P(i) satisfying oracle σ
$WL := get\_initial\_wl(i_0)$;
while $WL \neq \emptyset$ **do**
  select $(t, n_n, s_n) \in WL$;
  $WL := WL \backslash \{(t, n_n, s_n)\}$;
  $\Phi'_{t_n} := compute\_predicate(t, n_n)$;
  $verdict, m := ask\_solver(\Phi'_{t_n})$;
  if $verdict = SAT$ **then**
    $i = generate\_input(m)$;
    $t_i := P(i)$;
    if t_i *satisfies* σ **then**
      return i;
    end
    $(src_0, dst_0, target_0)..(src_n, dst_n, target_n) := get\_unexplored(t_i)$;
    compute $(score_0)..(score_n)$ where $score_j = DS(src_j, dst_j, target_j)$;
    $WL := WL \cup \{(t_i, src_0, (score_0, target_0))..(t_i, src_n, (score_n, target_n))\}$;
  end
end
return *Not Found*;

Algorithm 17 describes how the guided DSE works. It is a refinement of Algorithm 15, where the selection strategy is guided by a weight. First, an input seed i_0 generates a trace, and its unexplored edges initiate the working list WL using `get_initial_wl`. Then, `select` chooses the best trace t from the WL and its node n_n to invert according to the score s_n. `compute_predicate` computes the path predicate Φ'_{t_n}. The function *compute_predicate* is assumed to generate correct path predicates [Dav+16b] (following the Definition 11 in Section 7.1). Function `ask_solve` evaluates this path predicate and returns either an `UNSAT` *verdict* or a `SAT` *verdict* with a corresponding model m. *generate_input* transforms this model to a new input i, as introduced in Section 7.1.1. If the *trace* $P(i)$ does not trigger a Use-After-Free, checked by the Oracle σ, *get_unexplored* extracts the unexplored edges of the slice from the trace t_i generated by $P(i)$. Then DS ranks these edges and returns their appropriate score. Finally, the trace t_i with these branch nodes and their scores are added to the working list WL. The *target* is stored in WL with the score, to prioritize nodes according to it in the following order: *use*, *free*, *alloc*. The algorithm stops either when σ holds on the current path, or when there are no more paths to explore.

Correctness and completeness of the exploration If Algorithm 17 terminates and returns an input value i satisfying the Oracle σ, then i is a proof-of-concept triggering a Use-After-Free on program P. We say then that WS-Guided ensures *correctness* of the exploration, assuming that σ is correct. On the other hand, exploring all traces of the slice is not always possible in a finite time, so we must bound the exploration with a timeout or a fixed number of traces. In a similar way, we bound the number of loop iterations during the exploration. We consider then that WS-Guided does not ensure *completeness* of the exploration.

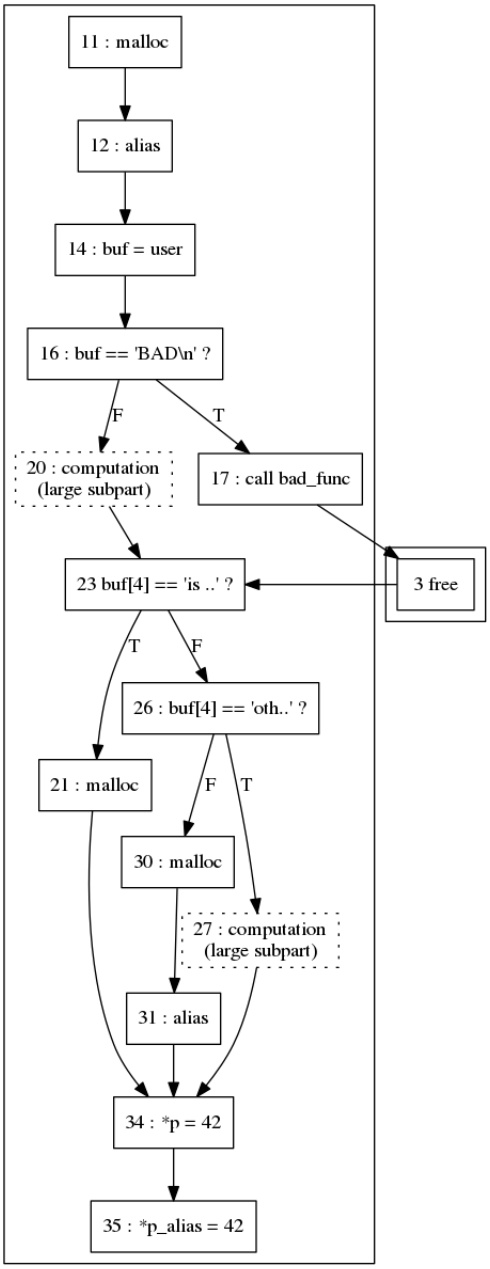
Motivating Example For the clarity of the explanation, Figure 7.5 recalls the motivating example introduced in Figure 3.2, and Table 7.1 recalls the Table 3.3 introduced in Chapter 3. As stated previously, in this table, the four different inputs represent four different path types (P_1 to P_4).

Input Example	Path Type	Use-After-Free ?
”AA..A”	P_1	no
”BAD\n”	P_4	no
”BAD\nis a uaf\n”	P_2	**yes** (36)
”BAD\nother uaf\n”	P_3	**yes** (35, 36)

Table 7.1: Generated inputs summary

```
1  void bad_func(int *p){
2   printf("This is bad, should exit !\n");
3   free(p);
4   // exit() is missing
5  }
6
7  void func(){
8   char buf[255];
9   int *p, *p_alias;
10
11  p=malloc(sizeof(int));
12  p_alias=p; // p_alias points to the same
        area as p
13
14  read(f,buf,255); // buf is user-controlled
15
16  if(strncmp(buf,"BAD\n",4) == 0){
17    bad_func(p);
18  }
19  else{
20    .. // some computation
21  }
22
23  if(strncmp(&buf[4],"is a uaf\n",9) == 0){
24    p=malloc(sizeof(int));
25  }
26  else if (strncmp(&buf[4],"other uaf\n",10)
        == 0){
27    .. // some computation
28  }
29  else{
30    p=malloc(sizeof(int));
31    p_alias=p;
32  }
33  // union of states
34
35  *p = 42 ; // is a uaf if line 16 and 26
        are true
36  *p_alias = 43 ; // is a uaf if line 16 and
        (23 or 26) are true
37 }
```

(a) Source code.



(b) Control Flow Graph.

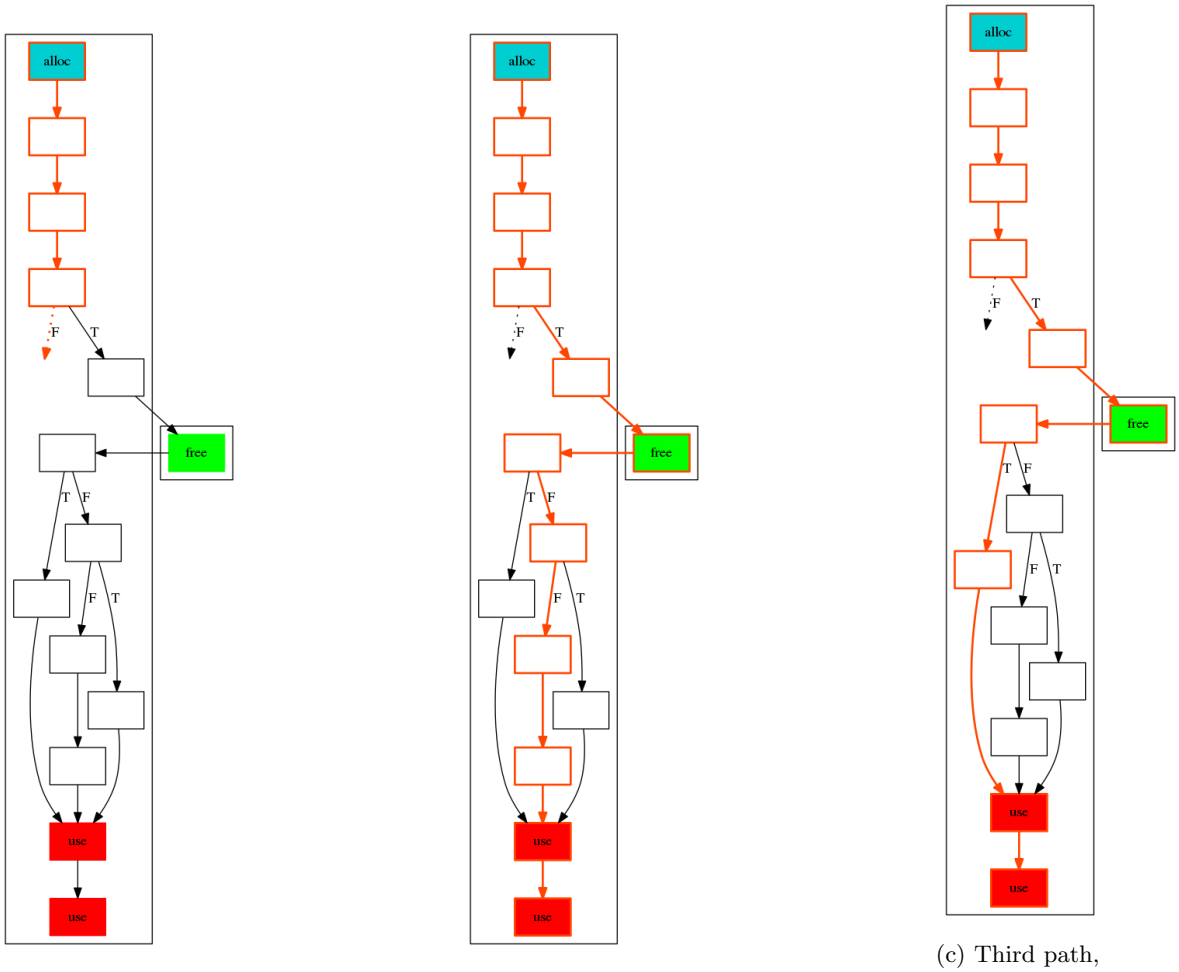
Figure 7.5: Motivating example (recalls Figure 3.2).

In our example, if we first try to explore the program with a file containing only ’A’[4] as

[4]This is a standard seed for dynamic analysis.

input, the first condition is evaluated to `true`, and we go out of the slice (see Figure 7.6a). Since the path goes out of the slice at the first condition, this one is selected for inversion. All the other possible conditions on this path are outside the slice and are thus not explored. The solver is able to solve the path predicate corresponding to the inversion of the selected condition and provides a new input starting with `"BAD\n"` (Figure 7.6b). However, condition 23 is still evaluated to `false` with this new input, and, in this case, there is no Use-After-Free, since the path belongs to P_4. DSE then produces a third input, by inverting line 23: `'BAD\nis a uaf\n'` (Figure 7.6c). As there is a Use-After-Free in the trace generated from this input, our Oracle detects it (see Section 7.3).

The exploration stops, and we now have a `proof-of-concept` *triggering the Use-After-Free.*



(a) First path, input filled with 'A'

(b) Second path, input: 'BAD\n'

(c) Third path, input: 'BAD\nis a uaf\n'

Figure 7.6: DSE: Traces generation (orange: current trace)

7.3 An Oracle Detecting Use-After-Free

As we introduced, DSE exploration needs an Oracle validating that a trace contains the desired property: triggering a Use-After-Free. This section presents our Oracle developed for this purpose. Such a validation is not a trivial task, as the state of the art of runtime detection demonstrates in Section 2.3.2. In particular, executing in sequence the three operations *alloc*, *free* and *use* on the same *address* is not a sufficient condition to trigger a Use-After-Free. As seen in Table 7.1, traces in P_4 fulfill this condition, but they correspond to false positives. We need *a way to validate if a trace contains a Use-After-Free.*

Issue The main problem when defining a *correct and complete runtime checker* for Use-After-Free detection is the aliasing due to re-allocations (*indirect aliases*), as shown in our motivating example. Allocation calls at lines 11 and 24 may return the *same* address when condition 16 is `true`. Therefore, pointers `p` and `p_alias` would be *indirect aliases*, but would not be coming from the same allocation call. In particular, in P_2, for which a simplification is provided in Listing 7.2, only `p_alias` is a dangling pointer and leads to a Use-After-Free if it is accessed. Tracking values is therefore not sufficient, and the allocation site of heap memory accesses has to be taken into account.

```
int *p, *p_alias;
p=malloc(sizeof(int));
p_alias=p; // p_alias points to the same area as p
free(p);
p=malloc(sizeof(int)); // p points to the same area as p_alias
*p = 42 ; // is not a uaf
*p_alias = 43 ; // is a uaf
```

Listing 7.2: Illustration of P_2 of the thesis motivating example.

Correctness and completeness In the following, the *correctness* of the checker corresponds to the absence of false positive (a Use-After-Free found is a true positive), and the *completeness* refers to its capacity to not miss Use-After-Free (all Use-After-Free are found).

7.3.1 Oracle Detecting Use-After-Free on a Trace

The Use-After-Free represented by the nodes N_{alloc}, N_{free} and N_{use} (recall Section 7.2.2) occurs on an execution trace t if and only if the following Property ϕ holds:

(i) $t = (\ldots, a_m := n_{alloc}(size_{alloc}), \ldots, n_{free}(a_f), \ldots, n_{use}(a_u))$, with $n_{alloc} \in^{\downarrow} N_{alloc}$, $n_{free} \in^{\downarrow} N_{free}$ and $n_{use} \in^{\downarrow} N_{use}$;

(ii) $a_m = a_f$ and the allocation site of a_f is n_{alloc};

(iii) $a_u \in [a_m, a_m + size_{alloc} - 1]$ and the allocation site of a_u is n_{alloc}.

Property ϕ could be verified using a classic data-flow analysis. In the following, we propose another approach directly based on the DSE and an original operation on path predicates. While DSE is classically used to test the feasibility of paths, we show here that it can have a more specific purpose.

The idea is to build a path predicate φ_t for the trace t, similar to a path predicate built for the exploration, except that all inputs are concretized, and we add one new symbolic variable S_{alloc} to represent a_m. Using a symbolic variable for S_{alloc} allows us to detect the absence of data-flow relations between this address and the ones used by n_{free} and n_{use}. It additionally avoids the problem of implicit aliases occurring with concrete values. We also keep track of the concrete value used as the size argument for this allocation, called $size_{malloc}$. Then, ϕ holds on a trace t if the SMT formula $\varphi_t \wedge \Phi'$ is `UNSAT`[5], where

$$\Phi' \triangleq (a_f \neq S_{alloc}) \vee (a_u \notin [S_{alloc}, S_{alloc} + size_{alloc} - 1]).$$

Φ' is the negation of the properties (ii) and (iii) of ϕ:

- $a_f \neq S_{alloc}$: the pointer given as the parameter for `free` is not the one allocated at n_{alloc} (negation of property (ii));
- $a_u \notin [S_{alloc}, S_{alloc} + size_{alloc} - 1]$: the allocation site of the pointer used is not n_{alloc} (negation of property (iii)).

If $\varphi_t \wedge \Phi'$ is `SAT`, at least one of the two properties is not met; thus the Use-After-Free is not present. However, if $\varphi_t \wedge \Phi'$ is `UNSAT`, these two conditions are false, and thus (ii) and (iii) are respected: a Use-After-Free is present.

$\varphi_t \wedge \Phi'$ is the Oracle σ in Algorithm 17. As (ii) and (iii) are respected, Use-After-Free detected by our Oracle are true positives.

[5]Notice that φ_t is always satisfiable.

Constraints on S_{alloc} If the value of S_{alloc} is let totally free, the solver could return a model making p pointing to the stack. Then writing to p would write to the stack and potentially modifying p itself or p_alias. To avoid such behavior, we bound the possible values of S_{alloc} to the addresses within the heap section.

Examples In the motivating example (Figure 3.2a), we have three types of traces containing n_{alloc}, n_{free} and n_{use}.

- For traces in P_2[6]:

$$\begin{aligned} \varphi_{t_{35}} &= (p\_0 = S_{alloc} \wedge p\_alias_0 = p\_0 \wedge p\_1 = 0x8040000) \\ \Phi'_{35} &= (p\_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_1 < S_{alloc} + 4) \end{aligned}$$

$\varphi_{t_{35}} \wedge \Phi'_{35}$ is `SAT` (e.g., with $S_{alloc} = 0x0$): there is no Use-After-Free for P_2 at line 35.

$$\begin{aligned} \varphi_{t_{36}} &= (p\_0 = S_{alloc} \wedge p\_alias_0 = p\_0 \wedge p\_1 = 0x8040000 \wedge *p\_1 = 0x42) \\ \Phi'_{36} &= (p\_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_alias_0 < S_{alloc} + 4) \end{aligned}$$

$\varphi_{t_{36}} \wedge \Phi'_{36}$ is `UNSAT`, which confirms the presence of the Use-After-Free at line 36.

- For traces in P_3:

$$\begin{aligned} \varphi_{t_{35}} &= (p\_0 = S_{alloc} \wedge p\_alias_0 = p\_0) \\ \Phi'_{35} &= (p\_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_0 < S_{alloc} + 4) \end{aligned}$$

$\varphi_{t_{36}} \wedge \Phi'_{36}$ is `UNSAT`, which confirms the presence of the Use-After-Free at line 35.

$$\begin{aligned} \varphi_{t_{36}} &= (p\_0 = S_{alloc} \wedge p\_alias_0 = p\_0 \wedge *p\_0 = 0x42) \\ \Phi'_{36} &= (p\_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_alias_0 < S_{alloc} + 4) \end{aligned}$$

$\varphi_{t_{36}} \wedge \Phi'_{36}$ is `UNSAT`, which confirms the presence of the Use-After-Free at line 36.

- For traces in P_4:

$$\begin{aligned} \varphi_{t_{35}} &= (p\_0 = S_{alloc} \wedge p\_alias_0 = p\_0 \wedge p\_1 = 0x8040000 \\ &\quad \wedge p\_alias_1 = p\_1) \\ \Phi'_{35} &= (p\_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_1 < S_{alloc} + 4) \end{aligned}$$

$\varphi_{t_{35}} \wedge \Phi'_{35}$ is `SAT` (e.g., with $S_{alloc} = 0x0$): there is no Use-After-Free for P_4 at line 35.

$$\begin{aligned} \varphi_{t_{36}} &= (p\_0 = S_{alloc} \wedge p\_alias_0 = p_0 \wedge p\_1 = 0x8040000 \\ &\quad \wedge p\_alias_1 = p\_1 \wedge *p\_1 = 0x42) \\ \Phi'_{36} &= (p\_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_alias_1 < S_{alloc} + 4) \end{aligned}$$

$\varphi_{t_{36}} \wedge \Phi'_{36}$ is `SAT` (e.g., with $S_{alloc} = 0x0$): there is no Use-After-Free for P_4 at line 36.

7.3.2 Correctness and Completeness of the Oracle

Correctness We said that our Oracle is correct; a Use-After-Free found is a true positive, there is no false positive.

Completeness over *malloc* return values Our approach does not work in the specific case of an array indexed by the value returned by *malloc*. Listing 7.3 provides such an example. Here, a Use-After-Free **can** occur if the value returned by *malloc* has its least significant bits equal to zero (computed at line 4). Our computation of allocation site does not work here, since the solver can find a solution for which the allocation site is different (and the Use-After-Free does not occur). While this example shows a limitation of our approach, it is highly specific; such a programming pattern should not appear in real-world programs. To the best of our knowledge, no other technique detecting Use-After-Free (detailed in the related work Section 2.3) considers such an example.

[6] In this example, malloc returned `0x8040000` as concrete value, and `sizeof(int)=4`. p_0 and p_1, p_alias_0 and p_alias_1 are the SSA variables created during the path predicate computation.

```
1  int *arr[256];
2  int index;
3  arr[0]=malloc(4);
4  index=arr[0]&0xff;
5  free(arr[0]);
6  arr[index][0]=42; // uaf if index is 0
```

Listing 7.3: Limitation example.

On the other hand, our Oracle is complete over a trace for Use-After-Free that are not dependent on the value returned by `malloc` (which should be, from a practical point of view, always the case); if the Oracle does not find the targeted Use-After-Free then this trace does not contain it.

Completeness over input values Our Oracle does not guarantee that a trace containing exactly the same instructions, but with different inputs, does not contain a Use-After-Free. Let us consider the example given in Listing 7.4. In this case, as our Oracle is based on the concrete values of the input (i is concretized), it can detect the Use-After-Free only if $i = 0$ during the execution.

```
1  int * buf[2];
2  int i=input() ;
3  buf[0]=malloc();
4  free(buf[0]) ;
5  buf[i][0] = 0; // is uaf if i=0
```

Listing 7.4: Data-dependent Use-After-Free.

Our trace exploration covers each trace at most once and does not try to also explore all possible behaviors coming from data. A solution to solve this problem would be to universally quantify each input using the $\forall$ quantifier. As detailed in Section 7.1.2, we cannot rely on such a quantifier in our formulas.

7.4 Guided DSE: Validity of Results

In this section, we discuss the validity of the results produced by the combination of the static analysis with the DSE technique. Table 7.2 summarizes this discussion; we are interested here in three aspects: the coverage (i.e., the set of paths analyzed), the validity of the Use-After-Free found and the Use-After-Free missed.

Paths analyzed In term of coverage, the static analysis results in:

- Feasible paths;
- Infeasible paths (due, for example, to the path-insensitivity, as detailed in Section 4.7.2);
- And some possible paths that are not analyzed (due, for example, to the bounded unrolling, as illustrated in Section 4.8).

Feasible and infeasible paths are represented through the slice. DSE works on concrete execution traces; all traces produced are then feasible. However, as our DSE exploration does not ensure completeness (see Section 7.2.5), it does not explore all the possible traces of the slice and thereby some paths are unexplored.

Statically detected Use-After-Free The static analysis presented in Part II produces false positives, true positives, and false negatives. On the other hand, thanks to the correctness of our Oracle, Use-After-Free found by the guided DSE are true positive, and there is no false positive. Thereby, the guided DSE does not produce any false positives. Yet, as the exploration is bounded, we do not ensure the absence of false negatives.

Use-After-Free not detected statically The DSE does not explore Use-After-Free that are missed by the static analysis; thereby, the exploration does not find Use-After-Free which are not statically detected.

Static Analysis	Results	Guided DSE	Results
Paths analyzed	Feasible, infeasible and unexplored paths	Concrete execution traces	All paths are feasible
		Bounded DSE exploration	Unexplored paths
Use-After-Free detected	True positives and False positives	Oracle on a trace	True positives and no false positives
		Bounded DSE exploration	False negatives
Use-After-Free not detected	False negatives	Not explored	False negatives

Table 7.2: Validity of the guided DSE results

Generally speaking, the static analysis aims to produce some true positives, with the smallest likelihood of them being false positives; yet, only with this analysis, the distinction between true and false positives is not possible. On the other hand, DSE does not produce any false positives; yet it would not be able to find Use-After-Free without the guidance of the static analysis. The strength therefore resides in the combination of the two methods.

Our analysis therefore produces only true positives and false negatives.

7.5 Guided DSE: Related Work

Guided DSE has been an area of active research in the past years. Several results follow the same concept as ours: first, a static step identifies interesting parts of the program, then a DSE explores these parts. We briefly describe some of these methods below.

In [ZC10], authors combine static analysis with DSE using a *proximity heuristic* computed statically. This heuristic guides the exploration to generate similar traces leading to a same bug. In his thesis, [Ma+11] proposes several heuristics to guide a DSE exploration toward a specific line of code. It is based on a mix between shortest path analysis and a backward analysis. In [Bab+11], a data-flow analysis is combined with shortest path in the Visible Pushdown Automaton (VPA) representation of the program to find vulnerabilities. We can also mention Dowser [Hal+13], which finds buffer-overflows by guiding the DSE tool S2E [CKC11] to focus on execution paths containing "complex" array access patterns, identified by a (lightweight) source-level static analysis. [MC13] combines shortest paths on conditional instructions with a data-flow analysis to remove unreachable paths. In [Li+13], authors used DSE to confirm memory leaks found in C++ programs by the static analyzer HP Fortify [HP]. Authors of [Che+14] first use a static analysis to check common classes for runtime errors in a C program, then they try to trigger all remaining potential errors through DSE restricted to a slice of the original program. In [Bar+14; Bar+15], the authors design a similar approach for proving the infeasibility of some white-box testing objectives, before launching DSE in order to cover them all. [Zha+15b] proposes to join a data-flow analysis with *Finite State Machine* (FSM) to select a path that satisfies a property as soon as possible.

If our work follows a similar approach, it differs in several respects: it fully operates on binary code, both on the static and dynamic side; the DSE is guided here by a weighted slice containing a set of (potentially) vulnerable paths. Moreover, these paths need to contain several targets in a specific order since we are looking for Use-After-Free. Finally, our algorithm is not restricted to a specific metric; we present our examples with shortest paths, yet any other metrics can be applied, as proposed in the next chapter.

7.6 Conclusion

This chapter defines our main contribution: how a slice produced by GUEB can guide the DSE. It details how this slice is weighted and presents a corresponding algorithm for exploration. Moreover, we provide an Oracle confirming the presence of Use-After-Free in a trace, with no false positives.

Chapter 8

Refining DSE Exploration

This chapter presents several proposals developed during this thesis to tackle the limitations of standard DSE when applied to real-world examples. DSE is an active research area, with many tools available, yet to be working, DSE engines need some specific tunings. We provide here the ones we found useful, improving either the path predicate computation or the guiding strategy.

Outline

- In the previous chapter, we have based the exploration on metrics to weight the slice. Section 8.1 provides an alternative to classic distance scores, based on random walks;
- Exploring library codes can be time-consuming during DSE; Section 8.2 proposes our solutions to tackle this problem;
- Section 8.3 presents a side contribution of this thesis: *C/S policies*;
- A classic problem during DSE exploration is handling the initial state; our solution is provided in Section 8.4;
- Finally, Section 8.5 concludes this chapter with a discussion of our contributions.

The first two sections provide improvements related to the DSE exploration, while the last two sections focus on improvements of the path predicate computation.

8.1 Distance Score Alternative: Random Walk

The notion of distance score used in guided DSE to drive the exploration towards a given target is classically based on shortest paths on the CFG (on basic blocks, instructions, conditional instruction, *etc*, see related work in Section 7.5). We propose in this section another metric, whose purpose is not to minimize the *number of instructions* to execute before reaching the target, but rather the *number of calls* to the solver, estimated by means of *random walks*. This section is an adaptation of our previous work [FMP16] to the particular case of guided DSE using GUEB.

First, we introduce the subject through a motivating example (not specific to Use-After-Free), then we formalize the usage of random walk to guide a DSE through a Use-After-Free.

8.1.1 Motivating Example

Figure 8.1 is used to illustrate our proposition. The right side of the figure is the control-flow graph representation of the source code, where node numbers correspond to source code line numbers. We assume that the objective of the DSE is to reach the call to function `goal` (line 11), and that `long_computation` (line 9) is a very large subgraph, with all internal paths leading to node 11. The slice to explore is represented by nodes: $\{3, 4, 9, 11\}$ (in the dotted square). Starting from node 3, there are three path categories:

(i) Paths that do not reach destination node 11 (and so leading outside of the slice) (called $Path_{out}$);

(ii) Paths that reach destination node 11 through node 4 (called $Path_{sp}$, paths in shortest paths);

(iii) Paths that reach destination node 11 through node 9 (called $Path_{lp}$, paths in longest paths).

Paths in $Path_{lp}$ are assumed to be significantly longer[1] than paths in $Path_{sp}$. Paths containing the node 4 can either be in $Path_{out}$ or $Path_{sp}$. Thus, going through node 4 can potentially lead to not reach the destination. Conversely, paths containing node 9 can only be in $Path_{lp}$, and choosing such paths will ensure to reach the destination.

```
1  void f()
2  {
3   if(..){
4    if(..){
5     go_out();
6    }
7   }
8   else{
9    long_computation;
10  }
11  goal();
12 }
```

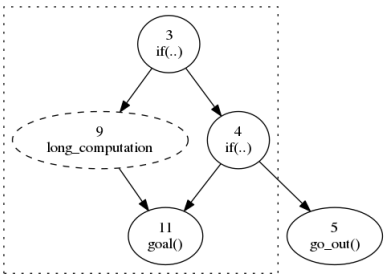


Figure 8.1: Random walk motivating example.

8.1.2 Proposition

We observe that most of the guided-DSE exploration strategies based on a distance use the notion of shortest path as a metric, without taking into account the context of program slice exploration. In Figure 8.1, state of the art strategies (presented in Section 7.5) will first select node 4 rather than node 9 to reach node 11 from node 3. However, this choice is clearly not the most appropriate. A path going through node 9 requires no further inversion of its path-predicate, while, passing through node 4, there is a 50% chance to need an inversion, which requires one more solver query. We notice that classic exploration of program slices do not take into account branches that do not reach the destination. For example, by removing node 5 and its incoming edge, the information needed to choose node 9 over 4 is lost. Our metric is based on this particular information. We propose a new path selection heuristic, based on the number of conditions to inverse necessary to reach the goal and hence, the number of solver queries, rather than on the length of a path. More specifically, we compute the probability[2] of a path starting from a node to reach a destination, while taking into account that it can go out of the slice.

8.1.3 Slice Creation

Our path selection heuristic is based on a variant of the slice representing Use-After-Free detailed in Chapter 5. Indeed, here we need to keep some edges that were removed from the original slice, as they add information for the guiding.

In a high-level perspective, from a Use-After-Free found by GUEB (represented here by its nodes N_{alloc}, N_{free}, and N_{use}), we create three slices. Each one is used to compute the distance to a targeted event, based on random walk. These distances are then used as score in the DSE exploration (in DS, Section 7.2.4).

Definitions reminder As reminded in Section 4.7.1, we note $G = (V, E)$, where G is the control-flow graph representation of the analyzed program. V is a set of nodes n_i and E a set of directed edges. $e_{i,j}$ denotes the edge between nodes n_i and n_j.

[1] In number of nodes.
[2] With respect to the CFG topology.

The guided DSE needs three $Weight_{target}$ functions, where the $target$ is either $alloc$, $free$, or use (see Section 7.2) N_{alloc}, N_{free} and N_{use} respectively account for the set of nodes representing allocation, free and use events of the Use-After-Free.

Slice creation From the original CFG, we create three slices, each one computing the score associated to one of the targets. First, we need to merge all the nodes of N_{target} to a specific new node: n_{target}. We then compute $G_{target} = (V_{target}, E_{target})$, the slice of G for computing $Weight_{target}$. V_{target} is the set of nodes reaching the node n_{target}. All nodes that do not reach n_{target} are merged into one single node: n_{out}. n_{out} and n_{target} are both absorbing nodes, meaning that we replace their out-edges by self-loops. Figure 8.2 defines the slice creation algorithm, where $reach(v_1, v_2)$ is $true$ if v_2 is reachable from v_1.

$$
\begin{aligned}
V_{target} = \quad & \{v \in V \mid reach(v, n_{target}) \wedge v \notin N_{target}\} \cup \{n_{target}\} \cup \{n_{out}\} \\
E_{target} = \quad & \{e_{i,j} \in E \mid n_i \in V_{target} \wedge n_j \in V_{target}\} \cup && (i) \\
& \{e_{i,target} \mid \exists e_{i,j} \in E \wedge n_i \in V_{target} \wedge n_j \in N_{target}\} \cup && (ii) \\
& \{e_{i,out} \mid \exists e_{i,j} \in E \wedge n_i \in V_{target} \wedge n_j \notin V_{target} \wedge n_j \notin N_{target}\} \cup && (iii) \\
& \{e_{target,target}\} \cup \{e_{out,out}\} && (iv)
\end{aligned}
$$

Figure 8.2: Slice creation.

The set E_{target} is split in four parts: (i) some edges of the original CFG kept in the slice, (ii) added edges to reach n_{target}, (iii) added edges to reach n_{out}, and finally (iv), added edges to make both n_{target} and n_{out} absorbing nodes.

Example To illustrate the slice creation, let us consider the example in Listing 8.1, which is a slight modification of the example in Figure 8.1. Figure 8.3a represents its CFG. In this example, there are two (similar) targets: n_8 and n_{13} (we illustrate here the fact that Use-After-Free can contain multiple *free* or *use* nodes). Figure 8.3b is the created slice. Nodes n_8 and n_{13} are replaced by n_{target}, and node n_5 by n_{out}. Edges defined by (ii) are in blue, while those coming from by (iii) are in red.

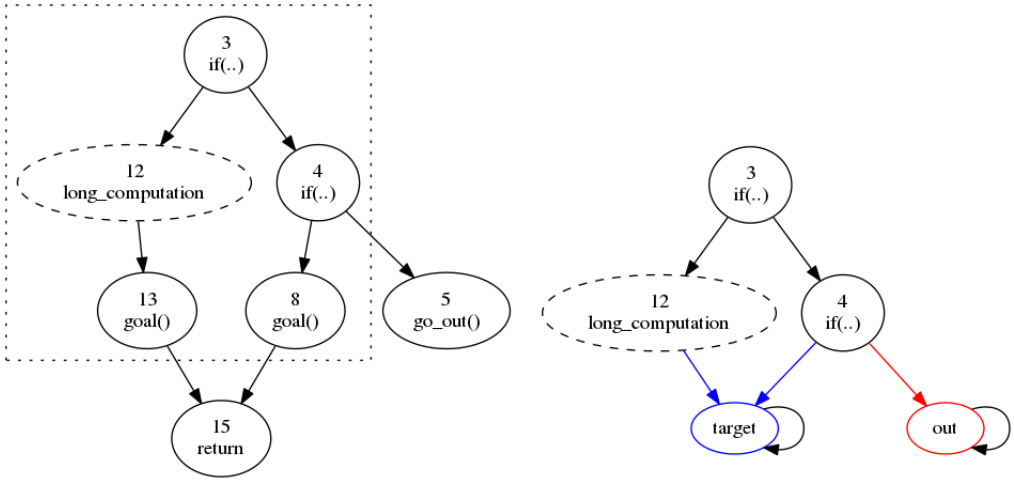
```
1  void f()
2  {
3   if(..){
4    if(..){
5     go_out();
6    }
7    else{
8     goal();
9    }
10  }
11  else{
12   long_computation;
13   goal();
14  }
15 return ;
16 }
```

Listing 8.1: Slice creation example.

8.1.4 Using Random Walk to Guide the Exploration

The main idea is now to compute, from a given node, the probability to reach rather n_{target} than n_{out}. This probability can be seen as the number of elementary paths[3] reaching n_{target} over the total number of elementary paths. Unfortunately, elementary paths computation is exponential. In order to provide a scalable solution, we propose to use a heuristic. A way to approximate program paths enumeration is to use random walks. The probability of executing a path going

[3]A elementary path is a path where no node appears more than once.



(a) Control-flow graph (b) Slice

Figure 8.3: Slice creation example from Listing 8.1.

from a given node to n_{target} is reduced to the probability for a random walk to reach n_{target} starting from n.

A random walk can be computed using the *transition matrix* of the graph [Lov93]. *A transition matrix* T is a $|N| * |N|$ matrix where $|N|$ is the number of nodes and $T(n_i, n_j)$ represents the probability for a random walk to move from n_i to n_j in one step. If there is no edge between n_i and n_j, this probability is 0, otherwise, it is equal to 1 divided by the number of out-edges of n_i in a unweighted graph:

$$T(n_i, n_j) = \begin{cases} 0 & \text{if } e_{i,j} \notin E, \\ \frac{1}{deg_{out}(n_i))} & \text{otherwise.} \end{cases}$$

Then $T^l(n_i, n_j)$ represents the probability of a random walk to be at n_j starting from n_i after l steps [Lov93].

Absorbing nodes An absorbing node a is a special node, where $T(a, a) = 1$ and $T(a, j) = 0, \forall j \neq a$. Absorbing nodes stop the random walk, since, once a path reaches an absorbing node, it can not move away from it. In a high-level perspective, $T^l(n_i, n_{abs})$, with n_{abs} an absorbing node, can be seen as the probability of a random walk to **reach** n_{abs} starting from n_i after l steps, rather than **be at** (meaning that the path does not necessary **terminate** at n_{abs}). From our slice creation (Section 8.1.3), n_{target} and n_{out} are both absorbing nodes.

Number of steps As we are interested only in the probability to reach absorbing nodes (and not the others), if the *stationary distribution* of T (i.e., the convergence of T^l) is too complex to compute, selecting simply a large number for l is sufficient. The intuition is that, as a path cannot move away from an absorbing node n_{abs}, $T^l(n, n_{abs}) \simeq T^{l+1}(n, n_{abs})$ for l large. From a practical perspective, raising a transition matrix to a large power is not a problem; our first experiment shows that, for a graph with 2000 nodes and 2600 edges, computing T^{200000} (l = 200,000) takes 8 seconds, which is clearly negligible compared to the computation time needed by a guided DSE to explore a program.

Example of transition matrix For our example in Listing 8.1 and FigureA8.3 we have:

$$T = \begin{array}{c} \\ n_3 \\ n_4 \\ n_{12} \\ n_{out} \\ n_{target} \end{array} \begin{array}{c} \begin{array}{ccccc} n_3 & n_4 & n_{12} & n_{out} & n_{target} \end{array} \\ \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

For example, $T(n_3, n_4) = \frac{1}{2}$, meaning that from n_3, and in one step, there is a $\frac{1}{2}$ chance to be at n_4. Then, the probability of a random walk to be at a node after 2 steps lies in:

$$T^2 = \begin{array}{c} \\ n_3 \\ n_4 \\ n_{12} \\ n_{out} \\ n_{target} \end{array} \begin{array}{c} \begin{array}{ccccc} n_3 & n_4 & n_{12} & n_{out} & n_{target} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & \frac{1}{4} & \frac{3}{4} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

For example, $T^2(n_3, n_4) = 0$, as after two steps, it is impossible to be at n_4, starting from n_3.

Score We can now define $Weight_{target}$, used by the Guided DSE (see Section 7.2) on random walk as follow:

$$Weight_{target}(n_{src}, n_{dst}) = T^l(n_{dst}, n_{target}).$$

Example of score In our example, $Weight_{target}(n_3, n_4) = T^2(n_4, n_{target}) = \frac{1}{2}$ and $Weight_{target}(n_3, n_{12}) = T^2(n_{12}, n_{target}) = 1$. Then, during the DSE exploration, between the edges $e_{3,4}$ and $e_{3,12}$, $e_{3,12}$ has a greater score than $e_{3,4}$, thereby $e_{3,12}$ is to explore at first.

8.1.5 Graph Pattern

```
1  if(cond)
2   small_calc() //
        small_calc contains
        paths that do not
        satisfy the desired
        property
3  else
4   long_calc() // in
        long_calc all paths
        satisfy the desired
        property
```

```
1  if(cond){
2   if(cond){
3           ... // the desired
                property is not
                satisfy
4   }
5   return ;
6  }
7  else if(cond){
8   return ;
9  }
10 return;
```

Figure 8.4: Pattern examples.

Our proposed strategy makes sense only if nodes in shortest paths could lead out of the slice. This corresponds to programming patterns shown in Figure 8.4. The first one is close to the example given in Figure 8.1. Here, the `true` branch of a comparison leads to a small function (in number of statements), but with paths inside this function that do not reach the target. On the contrary, the `false` branch leads to a larger function, with all paths reaching the target. The second pattern appears every time there are two comparisons, and only the first one contains another comparison leading paths to not reach the target. More generally, our approach differs from shortest path algorithms wherever the slice respects the following criterion:

Criterion 1. *In the slice, the path to n_{target} containing the minimal number of exit nodes is not a shortest path to n_{target}, where an exit node is a conditional node which may lead out of the slice.*

8.1.6 Evaluation and Perspectives

The next chapter provides benchmarks evaluating our guided DSE on real-world examples; however, we did not evaluate the use of random walks, other than in the example provided above. Indeed, we do not have Use-After-Free on a real-world program following the Criterion 1.

Nevertheless, this work direction is most promising. In particular, we believe that random walk could be mixed with other scores, such as classic shortest path or data-flow information. For example, as a random walk can be weighted, it would be interesting to use taint information to perform weighted random walk and base the guiding on this combination.

8.2 Exploration Tuning: Libraries

This section discusses how to tune the DSE exploration to the specific case of program libraries.

Libraries can be traced and added to the path predicate computation; yet this leads to an increase in the size of the path predicate, making it harder to solve. Moreover, it also leads to explore the code of libraries, which can be unnecessary. Libraries are well-known pieces of code, shared by different binaries. Our goal is, therefore, to take advantage of this property. We propose two different approaches:

- Either we locally improve the search heuristics by using a set of known code programming behaviors;
- Or we define stubs of libraries to model their effects without tracing them.

Our goal here is not to replace the main search heuristics, but rather to improve it locally to avoid getting stuck in uninteresting parts of the trace space.

8.2.1 Library-Driven Heuristics

Libraries are generally used will respect to some specific code programming behaviors. By knowing such patterns, we can improve the exploration and drive it with heuristics based on this information. We call these heuristics Libraries Driven Heuristics (LDH). They are meant to generate a set of traces that would be complicated to explore without specific guidance.

Exploration enhanced by LDH As we guide the DSE by using a score, we can detect when the exploration no longer progresses towards the destination (i.e., the score decreases). The idea is then to rely on LDH to generate a set of new paths to explore. After exploring the corresponding traces, and helped by the score, we can know if they are promising to reach the target destination.

LDH have to be defined by the user. In our current implementation, they are based on information about common libraries, such as the *libc* library. We describe in the following two LDH needed during the exploration of realistic codes.

String Length Based Heuristic

The first heuristic helps when there is a comparison of character strings whose lengths depend on the number of iterations of a previous loop. Let us consider this example:

```
1  read(f,tmp,255);
2  for(i=0;i<255;i++){
3    buf[i] = tmp[i];
4    if(tmp[i]=='\0') break;
5  )
6  buf[i]='\0';
7  if(strcmp(buf,"this is really bad") == 0)
8    ..
```

`strcmp` checks if the two strings passed as parameters are exactly the same. Every time the loop at line 2 is iterated, a constraint is added to the `path predicate` forcing `tmp[i]` (and so `buf[i]`) to be different from `'\0'`. If the seed input is an array of 255 `'A'`, the first path will unroll this loop 255 times. To evaluate the comparison at line 7, we need to unroll the loop at line 2 exactly 19 times (the size of the constant string plus the character `'\0'` which ends the string). This is a standard example where DSE can take time to explore a particular path. Indeed, in this case, DSE needs to explore all the 18 first iterations of the loop, before being able to invert line 7.

Our solution We propose to use the size $\mathbf{s_i}$ of constant strings passed to `strcmp` (and equivalent functions, as `strncmp`) to find this specific iteration number. We rely on this solution only to handle conditions located after such calls that we were not already able to invert.

In the previous example, the condition at line 7: .. == 0 follows a call to `strcmp`. We use the size of the string `"this is really bad"`: 19, to prioritize the inversion of conditions located at the 19^{th} iteration of the loop, saving us the exploration of conditions located in the 18 previous iterations. In the current implementation, all loops containing conditions in the iteration $\mathbf{s_i}$ are inverted. An improvement of this heuristic would be to combine it with a backward data-dependency analysis, to locate which loops have an impact on the parameters of the call to `strcmp`. Nevertheless, even with this naive implementation, this heuristic showa a real impact on DSE performance during the experiment performed on a CVE, provided in the next chapter.

Allocator Functions Behavior Based Heuristic

A second heuristic is imposed by the fact that some paths need specific results from allocator functions to be reached. This is illustrated in the following code:

```
1  p=malloc(size);
2  if(p==NULL)
3  {
4          // path to trigger
5  }
```

Although this example appears simple, standard DSE tools are not able to trigger the path, since to reach line 4, `malloc` has to returns 0. This can be done, for example, if the size given to `malloc` exceeds the available size of the heap. Without taking into account such behavior, the probability to trigger it is really low.

Our solution When meeting calls to allocator functions in traces, our exploration engine tries to apply a set of rules following the known behaviors of these functions. For example, it tries to create an input leading to a large value for the size given to *malloc*, or a size of 0 to *realloc*.

8.2.2 Stubs

The stub mechanism implemented in Binsec/SE allows to over-approximate or simulate logical effects of an untraced library call. This makes it possible to preserve symbolic execution soundness without having to execute the library code symbolically. Stubs are required to lower the trace size. Furthermore, library calls generally contain themselves system calls that would require some over-approximation, thus specifying the stub at library-level is a fair balance.

The implemented mechanism allows us to parameterize actions to be performed on the parameters and the result of a given function. Possible actions are

- `CONC`: concretize the given value (take runtime value);
- `LOGIC`: apply the logical operation specified in the stub;
- `SYMB`: symbolize the value, creating a new symbolic input.

`realloc` is an example of library code where the stub is not straightforward, but necessary. Tracing this call adds complex constraints on the *path predicate*, especially on the heap state. Skipping this function without taking into account its logical effect induces errors. Indeed, if the pointer returned by `realloc` is different from the pointer given as parameter, the values present in the original buffer are copied into the fresh buffer. By not taking into account this side-effect, results of the analysis become incorrect. The stub mechanism allows to handle this behavior by skipping the code of this function while keeping its logical effect.

In our experiments in the next chapter, we define logical stubs on fifteen significant functions (such as `strcpy`, `strchr`), while we simply concretize the return value of the other functions (such as `open`, `printf`).

Correctness Stubs need to be *well-defined* in order to keep the correctness of path predicates:

- `SYMB` has to be applied only on input functions;
- The logical effects produced by the stubs have to represent the correct behavior of the function.

Difference between LDH and stubs A function, *realloc* for example, can result in LDH and stubs. The difference lies in the fact that LDH tunes the exploration strategies, while stubs preserve the soundness of the generated path predicate without tracing the function.

8.2.3 Related Work

Different solutions have been proposed to tackle the problem of handling libraries. Stubs have been used by different other tools, such as KLEE [CDE08] or Mayhem [Cha+12]. S2E [CKC12] follows another idea: based on a *selective symbolic execution*, it can decide to explore symbolically only some parts of the code. A typical use-case is to explore only paths present in a targeted library and not on the original program. To the best of our knowledge, no tool uses an approach similar to the LDH; yet it brings an important improvement to the exploration of real-world code, as we will see in the next chapter.

8.3 Path Predicate Improvement: C/S Policies

Aside from the main stream of this thesis, we participated in the definition and design of so-called *C/S policies* [Dav+16b] to better parameterize a DSE. Although this is not a core contribution of this document, we summarize here the main elements of this work, since they are needed to understand better some of the choices discussed in the next section.

C/S Policies define rules to decide when to apply the process of concretization or symbolization operations on the memory elements during the path predicate construction. These rules are expressed through the language *CSml* [Dav+16b]. Listing 8.2 shows an example illustrating *C/S policies* and their impacts.

```
1  char key[]="ABC";
2
3  int f(int i){ // 0 <= i < 3
4          if(key[i] == 'B'){
5                  // dest to reach
6          }
7  }
```

Listing 8.2: Example dependent on C/S policies and initial memory state.

Consider `i` to be an input, with an associated constraint: $0 \leq i < 3$. A first execution with input $i = 0$ yields the trace:

$$t = n_3, n_4, n_6$$

At line 4, there is a memory read at the offset i of the array *key*. During the path predicate computation, there are two possibilities to handle such operation:

(i) Keeping i as symbolic; the solver can choose any possible values[4] for i and thus $key[i]$;

(ii) Concretize i; the solver uses the concrete execution value and the read is performed on $key[0]$.

Φ_{t_4} for (i) is (in bold the effect of n_4):

$$key[] = "ABC" \wedge 0 \leq i < 3 \wedge \mathbf{key[i]} \neq' \mathbf{B'}$$

Φ_{t_4} for (ii) is:

[4] Any values in the range $0 \leq i < 3$.

$$key[] = "ABC" \wedge 0 \leq i < 3 \wedge \mathbf{i} = \mathbf{0} \wedge \mathbf{key}[\mathbf{0}] \neq' \mathbf{B'}$$

Here, the last branching condition can only be successfully inverted for (i); thus by using (ii), we can not reach the desired destination. While (i) allows DSE to explore more paths than (ii), it comes at a cost for the solver; keeping load or store operations symbolic generates more complex path predicates; the time spent to solve it is then greater. Thereby, this is not suitable for exploring large program in a finite time.

In [Dav+16b], we describe four classic policies, present in most of the existing tools (more specific policies can, naturally, be defined by *C/S policies*):

- CC: loads and stores are concretized;
- CS: loads are concretized, while stores kept as logical;
- SC: loads are kept as logical, while stores are concretized;
- SS: loads and stores are kept as logical.

This example shows that *C/S policies* are useful to handle load and store operations in a configurable way, yet these rules have other applications. For example, with them, it is easy to concretize a part of the code belonging to libraries or complex functions (such as cryptographic functions) or to tune the path predicate for specific usages (i.e., to develop an Oracle).

Correctness of policies As detailed in [Dav+16a], *C/S policies* which employ only concretizations and do not symbolize non-input memory locations ensure correctness of path predicates, and are called *well-defined* policies.

8.4 Path Predicate Improvement: Initial Memory

Dealing with uninitialized memory is a common problem of DSE exploration. This section discusses this issue and provides our solution.

One of the difficulties in generating inputs using DSE is to be sure that the model given by the solver can be transformed into real inputs. This is not always the case because the model can contain constraints on parts of the memory that are not user-controllable. Such a situation arises typically when part of the memory in the trace is read before being written.

In the previous example given in Listing 8.2, the constraint $key = "ABC"$ needs to be present in the path predicate, while all values of key are not present in the trace t. If we consider a path predicate (using the SS policy) built without this constraint, we obtain this formula at line 4:

$$\Phi_{t_4} \triangleq 0 \leq i < 3 \wedge key[i] \neq' B'.$$

The inversion of the last condition leads then to:

$$\Phi'_{t_4} \triangleq 0 \leq i < 3 \wedge key[i] =' B'.$$

A solver can then give as a model satisfying this formula: $i = 2; key[2] =' B'$. However, it is not possible to create an input corresponding to this model as the value of key is not controllable by the user.

Constraints on uninitialized memory must, therefore, be added to the path predicate. There are four standard solutions:

- A first solution is to let a free (symbolic) uninitialized memory, but it comes at the price of correctness of the path predicates (correct in the sense that models of path predicate generate producible inputs), introducing false positives to the DSE exploration;
- A second solution is to have as initial state a snapshot of the initial concrete memory. However, it leads to a heavy overhead for the solver;
- A third solution is to let to the user the possibility of adding this initial state manually and handling himself the trade-off between efficiency and correction (this solution is used in [Dav+16a]). Yet this makes more complex the usage of the DSE, as it is not straightforward to define which initial values will be required during the exploration;
- The last solution is only applicable if the C/S Policy always concretizes read operations: it consists in concretizing every byte of the memory that is read before being written (for example, Triton [SST] follows this solution).

Our solution We consider here another approach, based directly on the model given as a result by the solver. The idea is:

- (i) We consider as *valid* only models containing constraints on symbolic variables corresponding to inputs;
- (ii) We refine the path predicate if it generates an invalid model.

We refine the path predicate by re-executing the trace and gathering the concrete values on the part of the initial memory for which the solver gave a valuation in the invalid model. We then add these values as constraints to the path predicate. We repeat this operation until the model given by the solver is valid (or become unsatisfiable).

Example of invalid models In the previous example, if the model returned by the solver is $i = 2; key[2] =' B'$ we consider it as invalid, since $key[2]$ is not an input, and re-execute the trace to gather the real value of $key[2]$ at the beginning of the trace. The new path predicate is then:

$$\mathbf{key[2]} =' \mathbf{C'} \wedge 0 \leq i < 3 \wedge key[i] =' B'.$$

The solver gives now as model: $i = 1; key[1] =' B$'. We gather then the real value of $key[1]$; the path predicate becomes:

$$\mathbf{key[1]} =' \mathbf{B'} \wedge \mathbf{key[2]} =' \mathbf{C'} \wedge 0 \leq i < 3 \wedge key[i] =' B'.$$

And, finally, the solver gives a valid model: $i = 1$, leading to the desired new input.

Notice that the solver could have given $i = 0; key[0] =' B'$ as the second model, which would have required one more iteration in the process.

Solving algorithm Algorithm 18 details the recursive function *Solve*, which performs the process described above. First, *ask_solver* solves the path predicate Φ. If the predicate is SAT, but the model contains constraints on the initial state, checked by *get_initial_addr*, the algorithm re-executes the original trace to gather the desired constraints, using *get_constraints_initial*. Then *Solve* is called recursively with these new constraints added to path predicate. *Solve* stops either if the formula is $UNSAT$, or if the generated model does not contain any more constraints on the initial state.

Algorithm 18: *Solve*

```
Solve(Φ)
    v, m = ask_solver(Φ)
    if verdict = UNSAT then
        return v, m
    end
    addr_set = get_initial_addr(m)
    if addr_set = {} then
        return v, m
    end
    c = get_constraints_initial(addr_set)
    return Solve(c ∧ Φ)
```

Bounded iteration As the memory of a program is finite, our approach always converges; yet this convergence can take time. To avoid to spend too much time in this process, the recursivity of *solve* is bounded; an alert is raised if this bound is reached and the DSE tries to explore another trace. Nevertheless, our preliminary experiment (provided in the next chapter) demonstrates that, from a practical point of view, this heuristic is a suitable solution.

$\forall$ quantifier Another approach would be to solve the formula for any possible initial memory state. Yet this would require the use of $\forall$ quantifiers, which is not possible in the current state of our DSE (as explained in Section 7.1.2).

8.5 Conclusion

This chapter provides several heuristics improving the DSE. We develop these heuristics either to enhance the path predicate computation or to improve the DSE exploration. More precisely, we tackle the problem of the initial memory state, the topic of handling libraries, and we propose a new metric to be used as a distance score for the DSE exploration.

Chapter 9

Guided DSE: Preliminary Results

This section presents the implementation of our guided DSE into BINSEC/SE platform and describes the application of our approach to produce an input triggering a Use-After-Free in `JasPer` (CVE-2015-5221, which was found by GUEB).

Outline

- First, Section 9.1 provides details on the implementation of the guided DSE;
- Section 9.2 presents the results of our approach on the thesis motivating example;
- Section 9.3 contains the study of our guided DSE on `JasPer`;
- A discussion of the current limitations of our experiment is discussed in Section 9.4;
- Finally, Section 9.5 concludes this part with a discussion on possible perspectives of our approach.

9.1 BINSEC/SE

Our guided DSE is implemented inside the BINSEC/SE platform. BINSEC/SE is the part of the open source Binsec platform [BIN] dedicated to symbolic execution. BINSEC/SE relies on the DBA intermediate representation [DB15]. The platform is composed of a pintool (written in *C++*), called *Pinsec*, which gathers execution trace, and a core engine, written in *Ocaml*, which performs the path predicate computation and the dedicated analyses. More information on the design of BINSEC/SE can be found in [Dav+16a].

WS-Guided implementation For the purpose of this thesis, we implemented into the BINSEC/SE platform a generic exploration mechanism, based on Ocaml functors, following the Algorithm 15 in Section 7.1.5. The WS-Guided algorithm (Algorithm 17 in Section 7.2.5) is then built on top of this mechanism. The criterion σ is also implemented as a functor, and then the exploration strategies can be combined with different Oracles. The current implementation includes the Use-After-Free Oracle and also a `buffer overflow` detector. Several exploration heuristics are implemented, including a classic *DFS* (depth-first search), either on the whole program or on a slice, and a directed search based on distance score, possibly enhanced with library driven heuristics. As the libraries driven heuristics (LDH) require a score to know if the generated paths are interesting, we could not combine them with classic *DFS*. Shortest paths are computed with the `igraph`[1] library.

9.2 Validation of the Global Approach

To validate our global approach, we first apply GUEB and BINSEC/SE on two test-cases: the thesis motivating example and an example of Use-After-Free occurring in a loop. The distance

[1] http://igraph.org/

score used in the following example is the length of shortest paths between basic blocks. Binaries and configuration files for this example are available at https://j-feist.com/thesis.

Motivating example Based on the slice produced by GUEB (and detailed in Chapter 6), BINSEC/SE triggers the Use-After-Free present in the binary version of our motivating example. The source code of the binary version is provided in Appendix A.

A first input has to be provided. We choose a seed file filled with 255 *A*. From it, the exploration produces two new inputs to trigger the Use-After-Free:

- First, it generates an input containing `"BAD\n"` ;
- Then, it triggers the Use-After-Free by generating the input `"BAD\notheruaf"` .

It is interesting to notice that our engine produces `"BAD\notheruaf"` instead of `"BAD\isuaf"`, as in the binary, the path leading to `"BAD\notheruaf"` comes first in term of shortest paths.

This example validates our *end-to-end* approach: from a binary containing Use-After-Free, we succeeded in first detecting it statically with GUEB and then in using the static information to guide our DSE to trigger the Use-After-Free in only a few iterations.

Use-After-Free in Loop Consider now the example in Listing 9.1, inspired by the example Figure 7.2 in Chapter 7, where three events of a Use-After-Free occur in a loop. Figure 9.1 gives the slice produced by GUEB. This example highlights our loop handling. Here, to trigger the Use-After-Free, the file *input* needs to start with the string *UAF*.

```
1  void main(){
2   char buf[255];
3   int f=open("input",O_RDONLY);
4   int n=5;
5   int *p=NULL;
6   int i;
7   read(f,buf,255); // buf is tainted
8
9   for(i=0;i<n;i++){
10   if(buf[0]=='U'){
11    p=malloc(sizeof(int));
12   }
13   if(buf[1]=='A'){
14    if(p) free(p);
15   }
16   if(buf[2]=='F'){
17    *p=0;
18   }
19  }
20 }
```

Listing 9.1: Use-After-Free in a loop.

From an initial seed file filled with 'A', our guided DSE succeeds in triggering the Use-After-Freein three iterations, as expected:

- First, it produces an input containing "U". No Use-After-Free occurs;
- Then, it produces an input containing "UA". No Use-After-Free occurs;
- Finally, the Use-After-Free is triggered, as the DSE produces the input "UAF".

This example validates our partitioning of traces, necessary for handling events present in a loop, as detailed in Chapter 7.

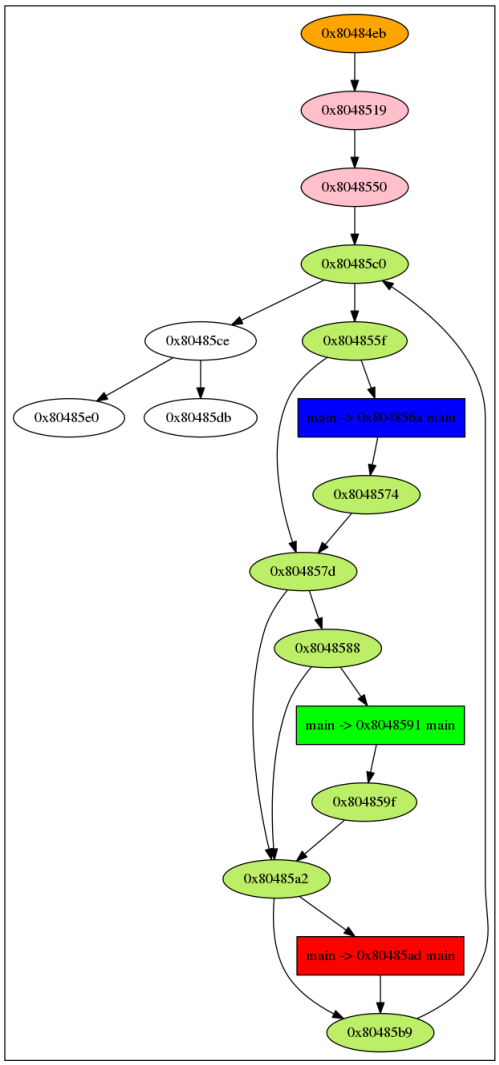


Figure 9.1: Slice produced by GUEB on the example in Listing 9.1.

9.3 JasPer: case study

We demonstrate the efficiency of our approach through the study of a Use-After-Free in `JasPer`[2]. `JasPer` can be used to convert an image from one format to another. As mentioned in Chapter 6, the source code of `JasPer` contains around 34,200 lines of *C* code. For this program, GUEB returns several suspicious slices. We detail in the following the validation of a particular selected slice, which, after a manual analysis, was strongly suspected to contain a Use-After-Free. The issue of the number of possible slices to explore is discussed in Section 9.4.

9.3.1 Vulnerability Description

The vulnerability was found by GUEB and is located in the function `mif_process_cmpt` (**CVE-2015-5221**). The entry point of the slice is the function `mif_hdr_get`. The first step of the analysis is therefore to manually find a way to trigger this function starting from the `main` function. Fortunately, this is directly done by forcing `JasPer` to take as input an image in the format `MIF` (which is a format specific to `JasPer`). The following command triggers `mif_hdr_get`:

```
1 jasper --input input --input-format mif --output output --
    output-format jpg
```

The file `input` is then converted from the format `MIF` to the format `JPEG` into the file *output*.

[2] https://www.ece.uvic.ca/~frodo/jasper/

Proof-of-concept Our DSE engine takes as input the weighted slice computed from GUEB, the command line and a first file *input* as seed file. In our experiments, we give a file filled with 'A'. The DSE successfully generates a test-case triggering the Use-After-Free (a double-free in this case), shown in Figure 9.2.

```
MIF
component
```

Figure 9.2: Proof-of-concept of CVE-2015-5221 generated by DSE.

This vulnerability is fixed in the latest version of `JasPer`[3].

Exploration We can separate the PoC in two parts: the first line `"MIF\n"` and the second line `"component"`. The first line is the result of a comparison byte per byte of the first four characters of the file, as shown in the following simplification of the code of `JasPer`:

```
if (m[0] != "M" || m[1] != "I" || m[2] != "F" || m[3] != "\n")
```

It is naturally easy for a DSE to solve this condition, and the first line is produced in only a few **iterations**. However, the second line, `"component"` is harder to generate. The following code represents a simplification of the necessary conditions to produce this line:

```
1  bufptr = buf;
2  while(i > 4096){
3          if ((c=get_char()) == EOF) break ;
4          *bufptr++=c;
5          i--;
6          if(c=='\n') break;
7  }
8  if (!(bufptr = strchr(buf,'\n'))) exit(0);
9  *bufptr = '\0'
10 ..
11 p = malloc();
12 strcpy(p,buf);
13 ..
14 if(!(strcmp(p,"component"))) exit(0);
```

The loop copies the input file in a buffer until it reaches the size of the buffer (4096) (line 2), or the end of the file (line 3), or the character `'\n'` (line 6). Then the function `strchr` and the next assignment replace the character `'\n'` in this buffer by `'\0'` (line 9). Finally, the buffer is copied to a new buffer and compared to the string `"component"` (line 12). Similarly to the example at Section 8.2.1, every time the loop is iterated, the constraint `buf[i] != '\n'` is added. Thereby, to successfully find an input validating the comparison made in `strcmp` (line 14), the comparison `c=='\n'` must be evaluated as `true` at the tenth iteration and false during the nine preceding iterations. In the code of `JasPer`, the loop and this comparison are distant in the code, so triggering this specific path is not straightforward.

9.3.2 Evaluation

We compare several versions of DSE and two standard fuzzers on their ability to detect the Use-After-Free in the considered slice. We report for each tool the time required to produce an input file with `"MIF\n"` as a first line and the time required to trigger the vulnerability. Experiments are performed on a standard laptop (`i7-2670QM`), and we use Boolector [NPB15b] as SMT solver[4]. Table 9.1 reports details of the experimentation. Here, LDH stands for library driven heuristics (Section 8.2) and DFS for depth-first search. In the following, we first comment the first four lines of this table, then the other four.

[3] https://github.com/mdadams/jasper/commit/df5d2867e8004e51e18b89865bc4aa69229227b3

[4] Other standard SMT solvers yield similar results; Boolector is given here for the sake of reproducible experiments.

Name	Time	MIF line	UAF found	# Paths
WS-Guided+LDH	$20m$	$3min$	**Yes**	9
WS-Guided	$6h$	$3min$	No	44
$DFS(slice)$	$6h$	$3min$	No	68
DFS	$6h$	$3min$	No	354
AFL	$7h$	$< 1min$	No	$174^{\dagger}$
$Radamsa$	$7h$	$> 1h$	No	$N/A^{\ddagger}$
AFL (MIF $input$)	$< 1min$	$< 1min$	**Yes**	< 10
$Radamsa$ (MIF $input$)	$< 1min$	$< 1min$	**Yes**	< 10

† *AFL generates more input, 174 is the number of unique paths.*
‡ *For radamsa it is not trivial to count the number of unique paths.*

Table 9.1: `JasPer` evaluation.

WS-Guided DSE and variants The generation of the PoC with the WS-Guided exploration (and LDH enabled) takes 20 minutes. Nine test cases are correctly generated, while 225 path predicates are `UNSAT`. The C/S policy [Dav+16b] used during our exploration kept as logical load addresses and concretized store addresses (SC). We limit the exploration of loops up to one hundred iterations. The size of the generated traces is about 10,000 instructions.

We try other variants of DSE in order to assess the effectiveness of our approach. In particular, we consider: WS-Guided DSE without LDH, standard DFS-based DSE and DFS-based DSE restricted to the slice of interest – where the search is cut when exiting the slice, but no distance score information is available (obtaining a technique similar to [Che+14], yet simpler). We bound the time of exploration of these variants to six hours. All these variants easily find the word `MIF`, but none of them is able to trigger the Use-After-Free, illustrating the importance of both weighted slices and LDH.

Comparison with fuzzing As a comparison with the state of the art technique, we used the `AFL` [Zal] (American Fuzzy Lop) and `radamsa` [Hel] tools to try to reproduce the crash. Starting from the same state, an input file filled with `'A'`, and the same command line, we run `AFL` and `radamsa` on `JasPer` for seven hours. We use both the simple and the `quick & dirty` mode on `AFL`, with the same results. *No crashes have been found.* These fuzzers are built in a perspective of coverage, not to find a particular path, thereby finding this specific bug is hard, as expected. We should mention that `AFL` generates an input with the `MIF` header in less than one minute, `radamsa` generates the word `MIF` after one hour and not as header of the file. The second line of the PoC being more complicated, they were not able to produce it.

Notice that by using as seed a correct `MIF` file (from the `JasPer` library), both fuzzers were able to generate the PoC in less than one minute. Since all proper `MIF` files contain the line `MIF` as header and at least one line starting with `component`, few mutations are necessary on these files to produce a PoC of the vulnerability.

Therefore, our approach is complementary to standard fuzzing techniques in certain situations, typically when no initial good seed is available. Moreover, recent tools such as Driller [Ste+16] or the so-called *cyber reasoning systems* (which are systems based on a combination of program analysis techniques) used during the Cyber Grand Challenge [Dar] showed that fuzzing can be efficiently combined with symbolic execution; such combinations are a natural evolution of our approach.

Initial memory On the experiment triggering the Use-After-Free (WS-Guided + LDH), constraints on the initial memory were produced by the heuristic detailed in Section 8.4. More precisely, five path predicates required constraints on the initial memory to be *correct* (i.e., solutions of path predicates yield producible inputs). For two of them, the process of adding

these constraints converged in one iteration, and the three others required two iterations. Four of these path predicates became unsatisfiable after the constraints were added. The last one allowed exploring a new trace.

This result validates the usefulness of our heuristic which gathers automatically and only when needed constraints on the initial memory. Without it, some parts of the program would not have been explored, as a path predicate was not initially correct and has resulted in an input which does not trigger the intended trace. Our first experiments showed that the convergence is fast; yet a more systematic and complete experiment is needed to validate the heuristic for real-world software.

9.4 Experimental Limitations

Slice selection The experiment on `JasPer` was applied on a slice that we knew to contain the vulnerability. A more systematic approach would apply BINSEC/SE to all the slices produced by GUEB. GUEB exports a few hundreds of different slices for `JasPer` (114 for $UafSet_{alloc}$ and 255 for $UafSet_{free}$, see Chapter 6). Having an infrastructure allowing the exploration of all these slices is realistic, but it was not possible in the context of this thesis. Moreover, there is a trade-off between what we call a *fully-automated analysis* (requiring no human intervention at all) and a *fully-human analysis* (where a human analyses one by one each slice produced by GUEB). We believe that according to available resources and time, a security researcher can first decide to discard some slices doing a manual analysis, and then explore only some of the slices with the DSE. Such *semi-automated analysis* is probably the best context for the application of our technique.

Moreover, even on a slice of which we knew to contain a Use-After-Free, automatically producing a proof-of-concept for it is definitely an desirable feature. Indeed, when a vulnerability is reported to the developer of the targeted program, being able to illustrate the vulnerability through an input is an effective way to demonstrate its dangerousness.

Experiments on other programs Unfortunately, we were not able to experiment our approach on real-world programs with Use-After-Free detected by GUEB other than `JasPer` (such as the one described in Chapter 6). This is mostly due to engineering constraints on the BINSEC/SE platform. This platform is still young, and the following limitations prevented us from performing other experiments:

- Some traces contained instructions that are not yet fully supported by the intermediate representation (DBA), making the path predicate computation impossible;
- The BINSEC/SE platform supports, for now, only inputs passing through *argv* and files; it is, for example, not possible to explore programs based on network socket (necessary, for example, to explore *accel-ppd*);
- Multi-threading applications are not on the scope of our DSE[5];
- BINSEC/SE does not manipulate efficiently traces containing too many instructions; the path predicate computation on large traces is not yet possible (it was the case for *openjpeg*).

9.5 Conclusion and Perspectives

Guiding DSE with information coming from a static analysis is a powerful combination. Our experiments showed that without this guiding, the vulnerability would not have been triggered in a reasonable amount of time. Moreover, they showed that DSE needs to be tuned to be applied on real-world examples. Enhancing DSE with knowledge from the user (such as using the libraries driven heuristics) helps a lot the exploration.

An interesting perspective would be to combine our DSE with fuzzers; their capacities to quickly generate several traces would increase the capacity of our guided DSE to reach the destination.

[5]Notice that GUEB can detect Use-After-Free in such a program if the vulnerability does not rely on thread properties, such as in *gnome-nettool*.

Another work direction is to test our approach on other CVE detected by GUEB. Although requiring more engineering work, it would allow to better evaluate how useful and generic are the heuristics presented in Chapter 8.

Finally, exploring with DSE other types of slice computed statically is an interesting work direction. For example, from a patched binary and its original version, binary comparison tools (such as BinDiff) allow us to know the difference between the two versions. It would be interesting to extract a slice from this difference and explore it with our approach. This would lead to the automated creation of `1day exploits` [Bru+08], which consists in exploiting unpatched programs.

Guided DSE: Summary of Contributions

This part presented the theoretical and practical aspects of the guided DSE. We recall here the contributions made:

- The description of how DSE is guided toward Use-After-Free using slices produced by GUEB (Chapter 7);
- The Oracle detecting Use-After-Free on a trace with no false positives and no false negatives;
- Several improvements on DSE, more precisely either on the path predicate computation and the guiding (Chapter 8);
- The demonstration of the efficiency of the *end-to-end* approach through the creation of a *proof-of-concept* on `JasPer`.

Part IV

Conclusion

Conclusion and Perspectives

In this thesis, we have studied how automated program analysis can be applied to trigger Use-After-Free on binary code. We have based our approach on the combination of two formal methods, static analysis and dynamic symbolic execution.

Program verification techniques applied to security A first objective of this thesis was to demonstrate that program verification techniques can be effectively applied to security purposes. Vulnerability discovery is an important topic for the *hacking* community, but, unfortunately, techniques and tools they used do not always benefit enough from the work and advances made in the academic community. We observe that one of the recurrent limitations preventing the use of formal methods is the difficulty to apply these methods to real-world programs. Conversely, this academic community is not sufficiently aware of the concrete problems raised by industrial code security analysts, and of the practical results obtained by some existing tools and techniques they used. Therefore, one particular effort of the present thesis was to try to overcome these limitations. The key insight of our approach was to couple formal methods with heuristics based on practical observations, allowing to scale to real-world programs. Therefore, we have put an important effort on manual inspection of our results to deduce suitable properties. As this thesis highlights, academic and industrial security researchers are two complementary communities, and the collaboration between them is, therefore, to be promoted and encouraged.

By adapting existing code analysis techniques to practical problems, we believe that this thesis partly answers one goal expressed in the introduction, which was to contribute to fill the gap between theory and practice in this domain.

Contributions Summary

We summarize here the contributions provided in this dissertation.

Use-After-Free: state of the art in detection Our first contribution, presented in Chapter 2, is to provide an overview of the state of the art concerning techniques and tools detecting or triggering Use-After-Free. To the best of our knowledge, it is the first attempt to explain in depth existing techniques (static analysis, dynamic analysis, and allocator) to analyze Use-After-Free.

Static analysis The second main contribution, which can be divided into several distinct steps, is the design of a static analyzer detecting Use-After-Free on binary code. Our work on this topic was directed by the lack of available and open solution to detect vulnerabilities on binary code statically. Thereby, the first step, presented in Chapter 4, was to design a memory model and a value analysis well suited for binary code. Several heuristics were developed, to make possible the analysis of real-world code. We studied the different ways to model the heap in Chapter 5. These models allow the formalization of the Use-After-Free detection. As our goal was the usability of our analysis, this Chapter also provided several Use-After-Free representations to supply suitable results for an analyst. All these theoretical aspects are validated in Chapter 6. Six previously unknown vulnerabilities were found through our open-source implementation, called GUEB. GUEB demonstrated to be robust enough to be used at large scale.

Dynamic symbolic execution The third main contribution lies in the results we obtained with dynamic symbolic execution. Our primary work on this technique is developed in Chapter 7: we have proposed an exploration algorithm using the information provided by the static analysis

to trigger Use-After-Free. Our algorithm ensures that, if it is able to produce an input triggering Use-After-Free, this Use-After-Free is real, and the input produced can demonstrate the existence of the vulnerability. This thesis also comes with several improvements for the dynamic symbolic execution technique itself, with a scope larger than Use-After-Free detection. We have proposed these improvements in Chapter 8, and showed how useful they are when exploring real-world binary codes and were needed for the validation of our benchmarks. Finally, the power of the coupling of static analysis with dynamic symbolic execution is demonstrated in Chapter 9, where it is used to produce an input triggering the CVE-2015-5221 found in the `JasPer` tool.

Publications and Community Contributions

This thesis resulted in several publications, talks, and contributions to the open-source community. The following academic papers and presentations are fully part of this dissertation:

- Josselin Feist, Laurent Mounier, and Marie-Laure Potet. "Statically detecting Use-after-Free on Binary Code". In: *Best Paper of GreHack 2013*. 2013 JCVHT ;
- Josselin Feist, Laurent Mounier, and Marie-Laure Potet. "Using static analysis to detect use-after-free on binary code". In: *1st Symposium on Digital Trust in Auvergne*. 2014 ;
- Josselin Feist, Laurent Mounier, and Marie-Laure Potet. "Guided Dynamic Symbolic Execution Using Subgraph Control-Flow Information." In: *SEFM*. ed. by Rocco De Nicola and eva Kühn. Lecture Notes in Computer Science. Springer, 2016 ;
- Josselin Feist, Laurent Mounier, Sébastien Bardin, and Robin David Marie-Laure Potet and. "Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free". In: *6th Software Security, Protection, and Reverse Engineering Workshop, SSPREW 2016, Los Angeles, CA, USA, December 5-6, 2016*. 2016 .

In addition, we participated to two other publications, which are evoked in this manuscript, but are not our main results:

- Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis." In: *SANER*. 2016 ;
- Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. "Specification of concretization and symbolization policies in symbolic execution." In: *ISSTA*. ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016 .

Moreover, part of this work has been presented to the hacking community through the following presentation:

- Josselin Feist. "GUEB : Static Detection of Use-After-Free on Binary". In: *ToorCon San Diego*. 2015 .

Finally, this thesis comes with several contributions to the open-source community:

- GUEB[Fei]: this thesis resulted in the built of a static analyzer detecting Use-After-Free on binary code (see Section II);
- Binsec/SE[BIN]: we contributed to the design and development of the open-source framework Binsec/SE (in particular for the symbolic execution, see Section III);
- Six unknown vulnerabilities were found using methods defined in this manuscripts in different open-source projects (see Chapter 6).

Perspectives

In addition to the perspectives already mentioned all along this manuscript, we can distinguish some primary work directions which could lead to interesting extensions of this work.

- As discussed in Section 6.6, GUEB could be improved in many manners; the most promising would probably be to build a second, more precise, but less scalable, memory model, used only on the part of the code where Use-After-Free have been detected. This second memory model could then focus on filtering numerous false positives.
- A more generic and complete test has to be performed on BINSEC/SE. Several tools are based on dynamic symbolic execution, but unfortunately, only a few of them have demonstrated to be efficient when applied to real-world code. While our approach has proven to work on a real CVE, we also lack of a more systematic benchmarks. The BINSEC platorm is a prototype that must be made more robust. Such a work would require a significant engineering effort, but it would provide a more complete validation schema.
- This thesis focused on Use-After-Free, yet other vulnerabilities are in the scope of our approach. The ones based on consequences of uninitialized values could be, for example, easily integrated into our tool-chain;
- The exploitability of Use-After-Free was not studied in this manuscript; possible outcomes of such a study could be the classification of Use-After-Free found or the prioritization of the exploration according to some exploitability goal. The study of the exploitability would require more work on the modeling of the allocation strategy, in particular, to deal with reallocations. Precise allocation strategy model is a topic understudied, and exploring such a subject is an attractive direction.

Bibliography

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling." In: *PLDI*. Ed. by Marina C. Chen, Ron K. Cytron, and A. Michael Berman. ACM, 1997.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. "Efficient Detection of All Pointer and Array Access Errors". In: 1994.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and algorithms (6.7 Strong Components)*. Addison Wesley, 1983.

[Akr10] Periklis Akritidis. "Cling: A Memory Allocator to Mitigate Dangling Pointers." In: *USENIX Security Symposium*. USENIX Association, 2010.

[Ana+16] Kapil Anand et al. "A Stack Memory Abstraction and Symbolic Analysis Framework for Executables." In: *ACM Trans. Softw. Eng. Methodol.* (2016).

[AS07] J. Afek and A. Sharabani. *Dangling pointer: POINTER. SMASHING THE POINTER FOR FUN AND PROFIT*. Black Hat USA. 2007.

[Bab+11] Domagoj Babic et al. "Statically-directed dynamic automated test generation." In: *ISSTA*. Ed. by Matthew B. Dwyer and Frank Tip. ACM, 2011.

[Bar+14] Sébastien Bardin et al. "An All-in-One Toolkit for Automated White-Box Testing." In: *TAP*. Ed. by Martina Seidl and Nikolai Tillmann. Lecture Notes in Computer Science. Springer, 2014.

[Bar+15] Sébastien Bardin et al. "Sound and Quasi-Complete Detection of Infeasible Test Requirements." In: *ICST*. IEEE Computer Society, 2015.

[Ber+00] Emery D. Berger et al. "Hoard: a scalable memory allocator for multithreaded applications". In: *SIGPLAN Not.* (2000). ISSN: 0362-1340.

[Bes+10] Al Bessey et al. "A few billion lines of code later: using static analysis to find bugs in the real world". In: *Communications of the ACM* (2010).

[BH13] Bernard Blackham and Gernot Heiser. "Sequoll: A framework for model checking binaries." In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2013.

[BR10] Gogul Balakrishnan and Thomas Reps. "WYSINWYX: What you see is not what you eXecute". In: *ACM Trans. Program. Lang. Syst.* (2010). ISSN: 0164-0925. URL: http://doi.acm.org/10.1145/1749608.1749612.

[Bru+08] David Brumley et al. "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications". In: 2008.

[Bru+11] David Brumley et al. "BAP: A Binary Analysis Platform". In: *Proceedings of the 23rd international conference on Computer aided verification*. CAV'11. Springer-Verlag, 2011. URL: http://dl.acm.org/citation.cfm?id=2032305.2032342.

[BZ06] Emery D. Berger and Benjamin G. Zorn. "DieHard: probabilistic memory safety for unsafe languages." In: *PLDI*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, July 17, 2006.

[BZ11] Derek Bruening and Qin Zhao. "Practical memory checking with Dr. Memory." In: *CGO*. IEEE Computer Society, 2011.

[Cab+12] Juan Caballero et al. “Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities.” In: *Proceedings of ISSTA*. Ed. by Mats Per Erik Heimdahl and Zhendong Su. ACM, 2012.

[CC77] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL ’77. ACM, 1977. URL: http://doi.acm.org/10.1145/512950.512973.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *OSDI*. 2008.

[Ces13] Silvio Cesare. “Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis”. In: *BlackHatUSA*. 2013.

[Cha+12] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP ’12. IEEE Computer Society, 2012. URL: http://dx.doi.org/10.1109/SP.2012.31.

[Che+14] Omar Chebaro et al. “Behind the scenes in SANTE: a combination of static and dynamic analyses.” In: *Autom. Softw. Eng.* (2014).

[Chr12] Veracode Christien Rioux. “Lessons In Static Binary Analysis”. In: *BlackHat US* (2012).

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: a platform for in-vivo multi-path analysis of software systems”. In: *ASPLOS*. 2011.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “The S2E Platform: Design, Implementation, and Applications”. In: *ACM Trans. Comput. Syst.* (2012).

[CR06] Sigmund Cherem and Radu Rugina. “Compile-time deallocation of individual objects.” In: *ISMM*. Ed. by Erez Petrank and J. Eliot B. Moss. ACM, 2006.

[CS13] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: *Commun. ACM* (2013). ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/2408776.2408795.

[CSB16] Xi Chen, Asia Slowinska, and Herbert Bos. “On the detection of custom memory allocators in C binaries.” In: *Empirical Software Engineering* (2016).

[CWEa] CWE. *CWE-415*. Double Free. URL: https://cwe.mitre.org/data/definitions/415.html.

[CWEb] CWE. *CWE-416*. Use After Free. URL: https://cwe.mitre.org/data/definitions/416.html.

[DA06] Dinakar Dhurjati and Vikram S. Adve. “Efficiently Detecting All Dangling Pointer Uses in Production Servers.” In: *DSN*. IEEE Computer Society, 2006.

[Dar] Darpa. *Cyber Grand Challenge*. https://www.cybergrandchallenge.com.

[Dav+16a] Robin David et al. “BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis.” In: *SANER*. 2016.

[Dav+16b] Robin David et al. “Specification of concretization and symbolization policies in symbolic execution.” In: *ISSTA*. Ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016.

[DB15] Adel Djoudi and Sébastien Bardin. “BINSEC: Binary Code Analysis with Low-Level Regions.” In: *TACAS*. Ed. by Christel Baier and Cesare Tinelli. Lecture Notes in Computer Science. Springer, 2015.

[DBG16] Adel Djoudi, Sebastien Bardin, and Eric Goubault. “Recovering High-Level Conditions from Binary Programs.” In: *FM*. Ed. by John S. Fitzgerald et al. Lecture Notes in Computer Science. 2016.

[DD06] Bruno Dutertre and Leonardo De Moura. “The yices smt solver”. In: *Tool paper at http://yices.csl.sri.com/tool-paper.pdf* (2006).

[DeM15] Jared DeMott. *Use-after-Free: New Protections, and how to Defeat them*. Bromium. https://labs.bromium.com/2015/01/17/use-after-free-new-protections-and-how-to-defeat-them/. 2015.

[Djo16] Adel Djoudi. "Binary-level static analysis". PhD thesis. 2016.

[Dol+16] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016.

[DP09] Thomas Dullien and Sebastian Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis". In: *CanSecWest* (2009).

[DRT15] David Dewey, Bradley Reaves, and Patrick Traynor. "Uncovering Use-After-Free Conditions in Compiled Code." In: *ARES*. IEEE Computer Society, 2015.

[Dut15] Bruno Dutertre. "Solving Exists/Forall Problems With Yices". In: *13th International Workshop on Satisfiability Modulo Theories (SMT 2015)*. 2015.

[EN08] Pär Emanuelsson and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools." In: *Electr. Notes Theor. Comput. Sci.* (Aug. 21, 2008).

[Eva06] Jason Evans. "A scalable concurrent malloc (3) implementation for FreeBSD". In: *Proc. of the BSDCan Conference, Ottawa, Canada*. 2006.

[Eva11] Jason Evans. *Scalable memory allocation using jemalloc*. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919. 2011.

[Fei+16] Josselin Feist et al. "Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free". In: *6th Software Security, Protection, and Reverse Engineering Workshop, SSPREW 2016, Los Angeles, CA, USA, December 5-6, 2016*. 2016.

[Fei15] Josselin Feist. "GUEB : Static Detection of Use-After-Free on Binary". In: *ToorCon San Diego*. 2015.

[Fer+11] Taís Borges Ferreira et al. "A Comparison of Memory Allocators for Multicore and Multithread Applications: A Quantitative Approach." In: *SBESC*. Ed. by Antônio Augusto Fröhlich and Leandro Buss Becker. IEEE Computer Society, 2011.

[Fer07] Justin N. Ferguson. *Understanding the heap by breaking it*. Black Hat USA. 2007.

[FFM12] T.B. Ferreira, M.A. Fernandes, and R. Matias. "A Comprehensive Complexity Analysis of User-Level Memory Allocator Algorithms". In: *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*. 2012.

[Fin] Jon Fingas. *Stagefright exploit reliably attacks Android phones*. https://www.engadget.com/2016/03/19/reliable-stagefright-android-exploit/.

[FMP14] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. "Using static analysis to detect use-after-free on binary code". In: *1st Symposium on Digital Trust in Auvergne*. 2014.

[FMP16] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. "Guided Dynamic Symbolic Execution Using Subgraph Control-Flow Information." In: *SEFM*. Ed. by Rocco De Nicola and eva Kühn. Lecture Notes in Computer Science. Springer, 2016.

[FMPHT] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. "Statically detecting Use-after-Free on Binary Code". In: *Best Paper of GreHack 2013*. 2013 JCVHT.

[Gar13] Tali Garsiel. *How browsers work : Behind the scenes of modern web browsers*. http://taligarsiel.com/Projects/howbrowserswork1.htm. 2013.

[Gio10] Vincenzo Iozzo Giovanni Gola. "Detecting aliased stale pointers via static analysis: An architecture independent practical application of pointer analysis and graph theory to find bugs in binary code". In: (2009-2010).

[git] git. *alloc.c - specialized allocator for internal objects*. https://github.com/git/git/blob/master/alloc.c.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *SIGPLAN Not.* (2005).

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “SAGE: Whitebox Fuzzing for Security Testing”. In: *ACM Queue* (2012).

[GMF06] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. “Free-Me: a static analysis for automatic individual object reclamation.” In: *PLDI*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006.

[God11] Patrice Godefroid. “Higher-order test generation.” In: *PLDI*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011.

[Gooa] Dan Goodin. *“Most serious” Linux privilege-escalation bug ever is under active exploit*. http://arstechnica.com/security/2016/10/most-serious-linux-privilege-escalation-bug-ever-is-under-active-exploit/.

[Goob] Peter Goodman. *PointsTo: Static Use-After-Free Detector for C/C++*. https://blog.trailofbits.com/2016/03/09/the-problem-with-dynamic-program-analysis/.

[Hal+13] Istvan Haller et al. “Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations”. In: *Proceedings of the 22Nd USENIX Conference on Security*. SEC’13. USENIX Association, 2013. URL: http://dl.acm.org/citation.cfm?id=2534766.2534772.

[Han90] David R. Hanson. “Fast Allocation and Deallocation of Memory Based on Object Lifetimes.” In: *Softw., Pract. Exper.* (1990).

[Hav97] Paul Havlak. “Nesting of Reducible and Irreducible Loops”. In: *ACM Trans. Program. Lang. Syst.* (1997). ISSN: 0164-0925.

[HB06] Matthew Hertz and Emery D. Berger. “Quantifying the performance of garbage collection vs. explicit memory management.” In: *OOPSLA*. Ed. by Ralph Johnson and Richard P. Gabriel. ACM, Feb. 13, 2006.

[Hee09] Sean Heelan. *Finding use-after-free bugs with static analysis*. https://sean.heelan.io/2009/11/30/finding-bugs-with-static-analysis/. 2009.

[HM05] Laune C. Harris and Barton P. Miller. “Practical analysis of stripped binary code.” In: *SIGARCH Computer Architecture News* (2005).

[INC12a] INCITS. *Information technology — Programming languages — C ISO/IEC 9899:2011*. 7.22.3 Memory management functions. 2012.

[INC12b] INCITS. *Information technology — Programming languages — C ISO/IEC 9899:2011*. 6.2.4 (2) Storage durations of objects. 2012.

[itsa] it-sec-catalog. *Analysis and exploitation (privileged)*. Object lifetime issues, Use-after-free, http://www.it-sec-catalog.info/privileged.html.

[itsb] it-sec-catalog. *Analysis and exploitation (unprivileged)*. Object lifetime issues, Use-after-free, http://www.it-sec-catalog.info/unprivileged.html.

[Kam98] Poul-Henning Kamp. “malloc(3) Revisited.” In: *USENIX Annual Technical Conference*. Ed. by Fred Douglis. USENIX Association, 1998.

[Kin10] Johannes Kinder. “Static analysis of x86 executables: = Statische Analyse von Programmen in x86”. PhD thesis. TU Darmstadt, 2010.

[Kin76] James C. King. “Symbolic execution and program testing”. In: *Commun. ACM* (1976). ISSN: 0001-0782.

[KK14] Vini Kanvar and Uday P. Khedker. “Heap Abstractions for Static Analysis.” In: *CoRR* (2014).

[Kor10] Sandeep Koranne. *Handbook of Open Source Tools*. 1st. 5 Apache Portable Runtime (apr) 5.1 APR Memory Pool. Springer-Verlag New York, Inc., 2010.

[Lan02] William Landi. “Undecidability of Static Analysis.” In: *LOPLAS* (Dec. 2, 2002).

[Lea00] Doug Lea. *A Memory Allocator*. http://g.oswego.edu/dl/html/malloc.html. 2000.

[Lee+15] Byoungyoung Lee et al. “Preventing Use-after-free with Dangling Pointers Nullification.” In: *NDSS*. The Internet Society, 2015.

[Li+13] Mengchen Li et al. “Dynamically validating static memory leak warnings.” In: *ISSTA*. Ed. by Mauro Pezzè and Mark Harman. ACM, 2013.

[Li10] Haifei Li. *Adobe Reader's Custom Memory Management: a Heap of Trouble*. BlackHat-EU. http://www.fortiguard.com/files/Adobe_Readers_Custom_Memory_Management_a_Heap_of_Trouble.pdf. 2010.

[Li15] Henry Li. *Microsoft Edge MemGC Internals*. hitcon 2015. https://hitcon.org/2015/CMT/download/day2-h-r1.pdf. 2015.

[Lov93] László Lovász. “Random walks on graphs: A survey”. In: *Combinatorics, Paul Erdos is Eighty* (1993).

[Ma+11] Kin-Keung Ma et al. “Directed Symbolic Execution.” In: *SAS*. Ed. by Eran Yahav. Lecture Notes in Computer Science. Springer, 2011.

[Mar08] Mark Marron. “Modeling the heap: A practical approach”. https://www.youtube.com/watch?v=AbiVYHVUOmQ. PhD thesis. Citeseer, 2008.

[MB08] Leonardo Mendonca de Moura and Nikolaj Bjorner. “Z3: An Efficient SMT Solver.” In: *TACAS*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Springer, 2008.

[MC13] Paul Dan Marinescu and Cristian Cadar. “KATCH: high-coverage testing of software patches.” In: *ESEC/SIGSOFT FSE*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013.

[McM] Robert McMillan. *How Heartbleed Broke the Internet — And Why It Can Happen Again*. https://www.wired.com/2014/04/heartbleedslesson/.

[Mic] Microsoft. *GFlags and PageHeap*. https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561.

[MM16] Xiaozhu Meng and Barton P. Miller. “Binary code is not easy.” In: *ISSTA*. Ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016.

[Mon16] David Monniaux. “A Survey of Satisfiability Modulo Theory.” In: *CoRR* (2016).

[Nag+09] Santosh Nagarakatte et al. “SoftBound: highly compatible and complete spatial memory safety for c.” In: *PLDI*. Ed. by Michael Hind and Amer Diwan. ACM, June 18, 2009.

[Nag+10] Santosh Nagarakatte et al. “CETS: compiler enforced temporal safety for C.” In: *ISMM*. Ed. by Jan Vitek and Doug Lea. ACM, 2010.

[NB10] Gene Novark and Emery D. Berger. “DieHarder: securing the heap”. In: *Proceedings of the 17th ACM conference on Computer and communications security*. CCS '10. ACM, 2010. URL: http://doi.acm.org.gate6.inist.fr/10.1145/1866307.1866371.

[ngi] nginx. *Memory Management API*. https://www.nginx.com/resources/wiki/extending/api/alloc/.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. “CCured: Type-safe Retrofitting of Legacy Code”. In: 2002.

[NMZ12] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. “Watchdog: Hardware for safe and secure manual memory management and full memory safety.” In: *ISCA*. IEEE Computer Society, 2012.

[NMZ14] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. “WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking.” In: *CGO*. Ed. by David R. Kaeli and Tipp Moseley. ACM, 2014.

[NPB15a] Aina Niemetz, Mathias Preiner, and Armin Biere. “Boolector 2.0 system description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* (2014 (published 2015)).

[NPB15b] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0 system description". In: *Journal on Satisfiability, Boolean Modeling and Computation* (2014 (published 2015)).

[NS07] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *PLDI*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, June 13, 2007.

[OCa] Robert O'Callahan. *Mitigating Dangling Pointer Bugs Using Frame Poisoning*. http://robert.ocallahan.org/2010/10/mitigating-dangling-pointer-bugs-using_15.html.

[Oli14] Albert Oliveras. "Survey of satisfiability modulo theories (SMT)". In: *Banff International Research Station for Mathematical Innovation and Discovery (BIRS) Workshop Lecture Videos*. Banff International Research Station for Mathematical Innovation and Discovery. 2014. URL: %5Curl%7Bhttp://www.birs.ca/events/2014/5-day-workshops/14w5101/videos/watch/201401221037-Oliveras.html%7D.

[OWA] OWASP. *Vulnerability*. https://www.owasp.org/index.php/Category:Vulnerability.

[Pro] ProFTPD. *Developer's Guide: Resource Pools*. http://www.castaglia.org/proftpd/doc/devel-guide/internals/pools.html.

[Roh] Chris Rohlf. *PartitionAlloc - A shallow dive and some rand*. https://struct.github.io/partition_alloc.html.

[RSI12] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows internals, 6th edition, part 2*. Pearson Education, 2012.

[SAJ16] Ahmed Saeed, Ali Ahmadinia, and Mike Just. "Tag-Protector: An Effective and Dynamic Detection of Out-of-bound Memory Accesses." In: *CS2@HiPEAC*. Ed. by Martin Palkovic et al. ACM, 2016.

[San] Paul Menage Sanjay Ghemawat. *TCMalloc : Thread-Caching Malloc*. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[Sch14] Edward J Schwartz. "Abstraction Recovery for Scalable Static Binary Analysis". PhD thesis. 2014.

[Ser+12] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX ATC 2012*. 2012. URL: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[Sha+03] Ran Shaham et al. "Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management." In: *SAS*. Ed. by Radhia Cousot. Lecture Notes in Computer Science. Springer, 2003.

[Sha87] Fred R Shapiro. "Etymology of the computer bug: History and folklore". In: *American Speech* (1987).

[Sil16] Natalie Silvanovich. *Life After the Isolated Heap*. https://googleprojectzero.blogspot.fr/2016/03/life-after-isolated-heap.html. 2016.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *SIGSOFT Softw. Eng. Notes* (2005).

[Son+08] Dawn Song et al. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*. 2008.

[spl15] sploitfun. *Understanding glibc malloc*. https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/. 2015.

[Sta] CERT-C Coding Standard. *MEM01-C. Store a new value in pointers immediately after free()*. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=440.

[Ste+16] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. The Internet Society, 2016.

[Sze+14] Laszlo Szekeres et al. “Eternal War in Memory.” In: *IEEE Security & Privacy* (2014).

[Tar74] Robert Endre Tarjan. “Testing Flow Graph Reducibility.” In: *J. Comput. Syst. Sci.* (1974).

[TL14] Bo Qu Tao Yan and Royce Lu. *Is It the Beginning of the End For Use-After-Free Exploitation?* Palo Alto Network. http://researchcenter.paloaltonetworks.com/2014/07/beginning-end-use-free-exploitation/. 2014.

[Ubu] Ubuntu. *Open bugs*. https://bugs.launchpad.net/ubuntu/+bugs.

[Val10] Chris Valasek. *Understanding the low fragmentation heap*. Blackhat USA. 2010.

[Vup] Vupen. *Technical Analysis of ProFTPD Response Pool Use-after-free (CVE-2011-4130)*. http://www.vupen.com/blog/20120110.Technical_Analysis_of_ProFTPD_Remote_Use_after_free_CVE-2011-4130_Part_I.php.

[Wan16] Wei Wang. “Partition Memory Models for Program Analysis”. PhD thesis. New York University, 2016.

[WHM13] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonca de Moura. “Efficiently solving quantified bit-vector formulas.” In: *Formal Methods in System Design* (2013).

[Wil+95] Paul R. Wilson et al. “Dynamic Storage Allocation: A Survey and Critical Review”. In: *IWMM*. Ed. by Henry G. Baker. Lecture Notes in Computer Science. Springer, 1995.

[XDS04] Wei Xu, Daniel C. DuVarney, and R. Sekar. “An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs”. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2004.

[Yai11] Michelle Yaiser. *Garbage collection internals for Flash Player and Adobe AIR*. http://www.adobe.com/devnet/actionscript/learning/as3-fundamentals/garbage-collection.html. 2011.

[Yas14] Mark Yason. *Understanding IE’s New Exploit Mitigations: The Memory Protector and the Isolated Heap*. Security Intelligence. https://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap/. 2014.

[Yas15] Mark Yason. *Understanding the Attack Surface and Attack Resilience of Project Spartan’s (Edge) New EdgeHTML Rendering Engine*. Black Hat US. https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdfLi,TrendMicro. 2015.

[You15] Yves Younan. “FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers.” In: *NDSS*. The Internet Society, 2015.

[ZC10] Cristian Zamfir and George Candea. “Execution synthesis: a technique for automated software debugging.” In: *EuroSys*. Ed. by Christine Morin and Gilles Muller. ACM, 2010.

[Zha+15a] Chao Zhang et al. “VTint: Protecting Virtual Function Tables’ Integrity.” In: *NDSS*. The Internet Society, 2015.

[Zha+15b] Yufeng Zhang et al. “Regular Property Guided Dynamic Symbolic Execution.” In: *ICSE (1)*. Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE, 2015.

Tools References

[35] Vector 35. *Binary Ninja*. https://binary.ninja/.

[BIN] BINSEC. *BINSEC/SE*. http://binsec.gforge.inria.fr/tools.

[Bit] Trail of Bits. *DARPA Challenge Binaries on Linux and OS X*. https://github.com/trailofbits/cb-multios.

[Cis] Talos Group Cisco. *FreeSentry*. https://github.com/yyounan/freesentry.

[Cla] Clang. *Clang Static Analyzer*. http://clang-analyzer.llvm.org/.

[Cov] Coverity. https://www.coverity.com/.

[Die] DieHarder. https://github.com/emeryberger/DieHard.

[DrM] Dr.Memory. http://www.drmemory.org/.

[Fei] Josselin Feist. *GUEB*. https://github.com/montyly/gueb.

[GCB] Gustavo Grieco, Martin Ceresa, and Pablo Buiras. *QuickFuzz*. http://quickfuzz.org/.

[Gooc] Google. *AddressSanitizer*. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[Good] Google. *BinNavi*. https://github.com/google/binnavi.

[Gooe] Google/Zynamics. *The MonoREIL static code analysis framework*. https://www.zynamics.com/binnavi/manual/html/mono_reil.html.

[Graa] GrammaTech. *CodeSonar*. https://www.grammatech.com/products/codesonar.

[Grab] GrammaTech. *CodeSonar Binary Code Analysis*. https://www.grammatech.com/products/binary-analysis.

[Han] M. Weiser Hans-J. Boehm Alan J. Demers. *A garbage collector for C and C++*. http://www.hboehm.info/gc/.

[Hel] Aki Helin. *Radamsa*. https://github.com/aoh/radamsa.

[Hex] Hex-Rays. *IDA*. https://www.hex-rays.com/products/ida/.

[HP] HP. *Fortify Static Code Analyzer*. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[LLV] LLVM. *The LLVM Compiler Infrastructure*. http://llvm.org/.

[Mat] MathWorks. *Polyspace Bug Finder*. https://mathworks.com/products/polyspace-bug-finder/.

[Nag] Santosh Nagarakatte. *SoftBoundCETS*. http://www.cs.rutgers.edu/~santosh.nagarakatte/softbound/.

[Per] Bruce Perens. *Electric Fence*. http://elinux.org/Electric_Fence.

[rad] radare2. http://www.radare.org/r/.

[SST] Jonathan Salwan, Florent Saudel, and Romain Thomas. *Triton*. https://triton.quarkslab.com/.

[Val] Valgrind. *Memcheck*. http://valgrind.org/info/tools.html.

[Vera] Veracode. *White Box Testing (SAST)*. http://www.veracode.com/products/binary-static-analysis-sast.

[Verb] Inc Veracode. *Veracode*. https://www.veracode.com/.

[Zal] Michal Zalewski. *AFL (american fuzzy lop)*. http://lcamtuf.coredump.cx/afl/.

Appendices

Appendix A

Motivating Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8
9  int f;
10 int a;
11
12 void bad_func(int *p){
13  printf("This is bad, should exit !\n");
14  free(p);
15  // exit() is missing
16 }
17
18 void func(){
19  char buf[255];
20  int *p, *p_alias;
21
22  p=malloc(sizeof(int));
23  p_alias=p; // p_alias points to the same area as p
24
25  read(f,buf,255); // buf is tainted
26
27  if(strncmp(buf,"BAD\n",4) == 0){
28    bad_func(p);
29  }
30  else{
31    a=4;
32  }
33
34  if(strncmp(&buf[4],"is a uaf\n",9) == 0){
35    p=malloc(sizeof(int));
36  }
37  else if (strncmp(&buf[4],"other uaf\n",10) == 0){
38    a=6;
39  }
40  else{
41    p=malloc(sizeof(int));
42    p_alias=p;
43  }
```

```
44   // merging state
45
46   *p = 42 ; // is a uaf if line 27 and 37 are true
47   *p_alias = 43 ; // is a uaf if line 27 and (34 or 37) are true
48 }
49
50 void main()
51 {
52           f=open("input",O_RDONLY);
53           func();
54 }
```

Listing A.1: Motivating example with main

Appendix B

Gnome-nettool

dabord inf nic changed, dct, slice trop gros. on constate que tout commence a info get nic information.

```
gueb -reil gnome-nettool-00 -func info_nic_changed -output-dir
    results/ -flow-graph-dot -flow-graph-call-disjoint
```

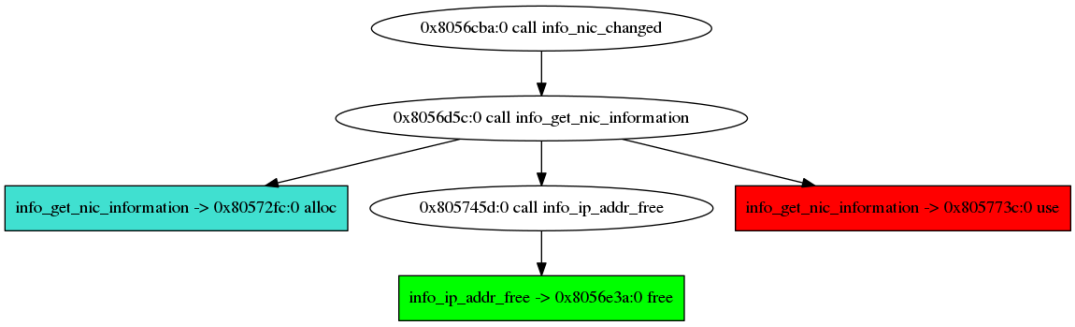


Figure B.1: *DCT*of the Use-After-Free found in the function info_nic_changed

```
gueb -reil gnome-nettool-00 -func info_get_nic_information -
    output-dir results/ -flow-graph-dot -flow-graph-call-
    disjoint
```

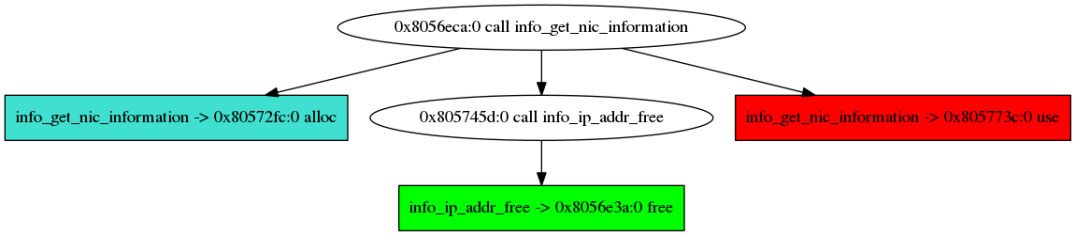


Figure B.2: *DCT*of the Use-After-Free found in the function info_get_nic_information

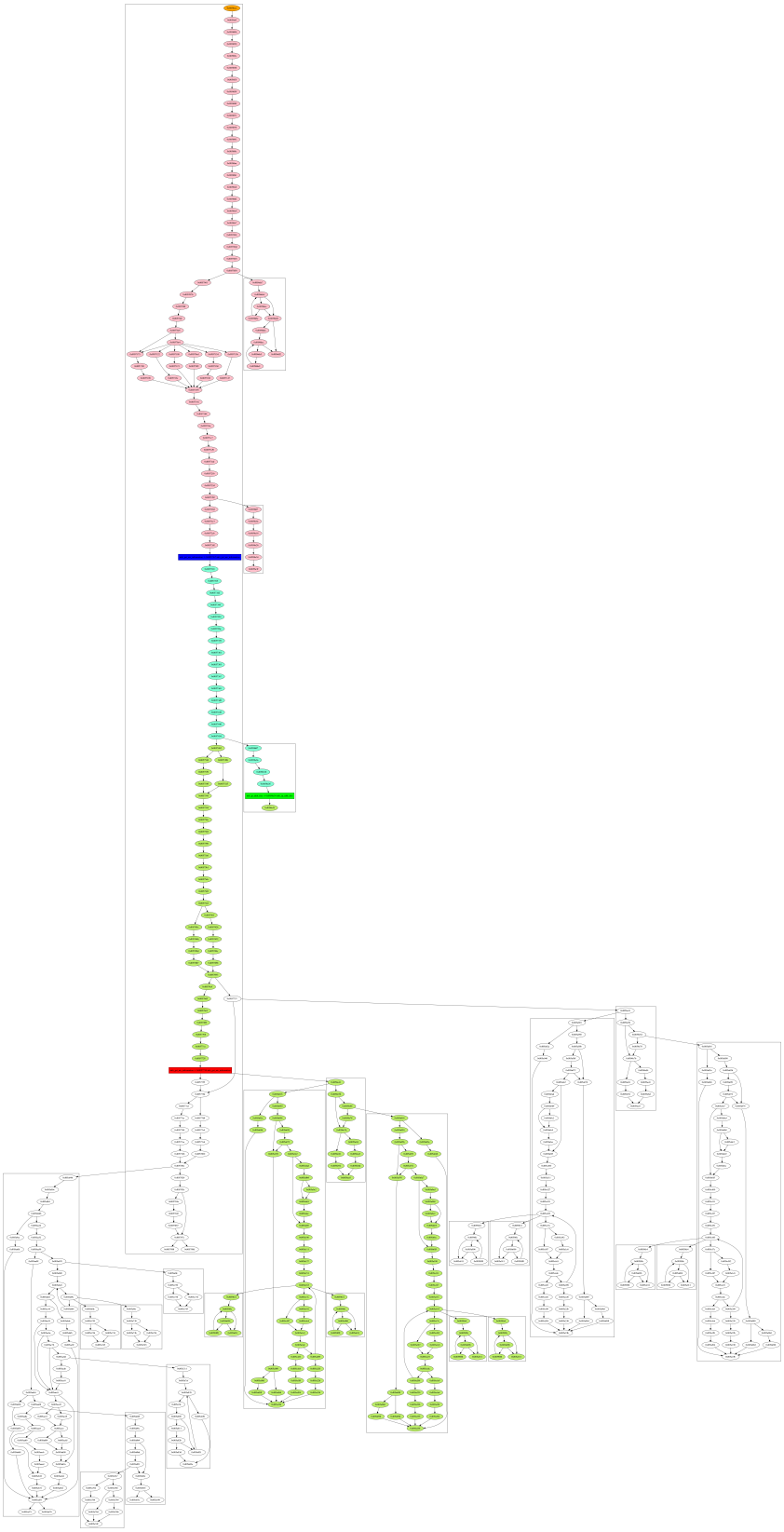


Figure B.3: Slice of the Use-After-Free found in the function info_get_nic_information

Appendix C

Gnome-nettool: stub

```
1  module StubGnomeNettool = functor (Absenv_v : AbsEnvGenerique)
       ->
2  struct
3      let gtk_tree_model_get  = 0x0804DBD0 ;; (* The address of
           the function *)
4
5      (* This function will restore esp *)
6      let restore_esp vsa = Absenv_v.restore_esp vsa;;
7
8      let call_gtk_tree_model_get vsa addr func_name call_number
           backtrack =
9          try
10             let vsa = restore_esp vsa in
11             let new_state = ((addr,func_name,call_number)::
                   backtrack) in
12             (* add malloc in the third argument *)
13             let vsa = Absenv_v.malloc_arg vsa new_state 36 in
14             true,vsa
15         with
16             _ ->
17                 true,restore_esp vsa
18
19     let stub addr_call vsa addr func_name call_number
           _backtrack  =
20         if (addr_call = gtk_tree_model_get) then
               call_gtk_tree_model_get vsa  addr func_name
               call_number _backtrack
21         else false,vsa
22
23 end;;
```

Listing C.1: Motivating example with main