POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

MemTrace: a dynamic memory overlaps tracing tool

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Kristopher Francesco Pellizzi**

Student ID: 920757
Advisor: Prof. Mario Polino
Co-advisors: Prof. Michele Carminati, Prof. Stefano Zanero
Academic Year: 2020-21

Abstract

The mitigation techniques employed by modern compilers and operating systems make it more difficult to exploit vulnerabilities lying in a program. However, they can be usually bypassed by simply leaking an address or some specific value stored in memory (e.g., a stack canary). A very frequent method used to get a leak is exploiting an uninitialized read vulnerability lying in the program. Indeed, stack and heap memory are often allocated and deallocated according to the program's needs and therefore the same memory location can be re-used multiple times during program's execution. This generates unintended overlaps in memory that, thanks to an uninitialized read, can be leveraged to obtain a leak. A manual analysis to discover uninitialized reads in a binary and understand if and how they can be exploited to obtain a leak is quite difficult and certainly requires a large amount of time. So, we designed *MemTrace*, a dynamic instrumentation tool that keeps track of the memory accesses performed by the executable and reports all the detected memory overlaps. To keep track of the state of bytes in memory, *MemTrace* uses a shadow memory that reflects the state (either initialized or uninitialized) of all the bytes of the stack and the heap; while in order to keep track of transfers of uninitialized bytes and to be able to detect the usage of uninitialized bytes, *MemTrace* implements a taint analysis, that marks as *tainted* a byte which is read while in the *uninitialized* state. By analyzing the report generated by *MemTrace*, the user can therefore detect potential leaking instructions together with the origin of the bytes it allows to read. To allow *MemTrace* to explore more paths of a program, we then paired it with a fuzzer in order to perform an automatic analysis of the binary. We tested *MemTrace* with 12 binaries containing a known uninitialized read vulnerability. When the binaries were executed with a crafted input triggering the vulnerability, *MemTrace* was always able to detect it; while when used in combination with the fuzzer, *MemTrace* was able to automatically detect the vulnerability for 8 of the tested binaries. In all the cases, *MemTrace* correctly reported the memory overlaps, thus proving that the used approach is effective.

Keywords: dynamic analysis, instrumentation, memory overlaps, uninitialized read, leaks, vulnerability

Abstract in lingua italiana

Le tecniche di mitigazione impiegate dai moderni compilatori e sistemi operativi rendono più difficile sfruttare le vulnerabilità che si trovano in un programma. Tuttavia, di solito possono essere aggirati tramite il leak di un indirizzo o di un valore specifico salvato in memoria (ad esempio uno stack canary). Un metodo molto frequente utilizzato per ottenere un leak è sfruttare una lettura non inizializzata che si trova nel programma. Infatti, stack e heap vengono spesso allocati e deallocati in base alle esigenze del programma e quindi la stessa area di memoria può essere riutilizzata più volte durante l'esecuzione. Ciò genera degli overlap di memoria che, grazie a una lettura non inizializzata, possono essere sfruttati per ottenere un leak. Un'analisi manuale per scoprire le letture non inizializzate in un binario e capire se e come possano essere sfruttate è difficile e richiede una grande quantità di tempo. Dunque, abbiamo sviluppato *MemTrace*, un tool di strumentazione dinamica che tiene traccia degli accessi alla memoria eseguiti dal programma e segnala tutti gli overlap di memoria rilevati. Per tenere traccia dello stato della memoria, *MemTrace* utilizza una shadow memory che riflette lo stato di tutti i byte dello stack e dello heap; mentre per tenere traccia dei trasferimenti di byte non inizializzati ed essere in grado di rilevare l'utilizzo di tali byte, *MemTrace* implementa una taint analysis, che contrassegna come *tainted* un byte che viene letto mentre si trova in uno stato non inizializzato. Analizzando il report generato da *MemTrace*, l'utente può quindi rilevare potenziali leak e verificare l'origine dei byte che quest'ultimo consente di leggere. Per permettere a *MemTrace* di esplorare più percorsi di un programma, lo abbiamo combinato ad un fuzzer. Abbiamo testato *MemTrace* con 12 binari contenenti una lettura non inizializzata nota. Quando i binari sono stati eseguiti con degli input in grado di triggerare la vulnerabilità, *MemTrace* è sempre stato in grado di identificarla; mentre quando è stato usato in combinazione con il fuzzer, *MemTrace* è riuscito ad identificare la vulnerabilità per 8 dei binari testati. In tutti i casi, *MemTrace* ha riportato correttamente gli overlap di memoria, dimostrando che l'approccio utilizzato è efficace.

Parole chiave: analisi dinamica, strumentazione, overlap di memoria, letture non inizializzate, leaks, vulnerabilità

Contents

Introduction

Nowadays mitigation techniques against memory corruption vulnerabilities are commonly implemented in essentially all the major operating systems and compilers. Techniques such as *stack canaries*, $W \oplus X$ and *ASLR* effectively raised the bar, thus increasing the effort needed to write an exploit that allows to inject and execute arbitrary code, as it usually requires a **leak** that allows the attacker to get either the stored *canary* or the address of some memory section or library function. A frequently used technique to get a leak consists in exploiting **uninitialized memory reads**. Since memory is a limited resource, it is continuously allocated and deallocated during a program's execution according to its needs. This way, memory locations can be reused multiple times, thus possibly generating some **unintended overlaps**. The combination of memory overlaps and uninitialized memory reads may therefore allow obtaining a leak that may be used to bypass mitigation techniques. Finding uninitialized reads in an executable and understanding if and how they can be exploited to obtain a leak is a very difficult and time-consuming task which may require **hours**, if not **days**, of analysis. This is especially true if we are dealing with a big, complex program which may even use many external libraries.

Some of the existing tools allow analyzing a binary looking for possible uninitialized reads ([36]). They, however, provide no insight information about what we can read from them, thus leaving the responsibility to perform additional manual analysis to the user, which must run the binary providing inputs and trying to figure out what is the origin of bytes read through the uninitialized read. Even this may not be an easy task. Indeed, it is possible that, due to different paths of execution triggered by different inputs, the bytes read through the same uninitialized read come from different origins. It is even possible that these bytes have been written by some instruction from another function executed much earlier than the execution of the uninitialized read or that they have been written by more than one instruction only partially filling a certain memory location, making the analysis even more difficult. Other approaches to try and detect uninitialized reads or leaks require the availability of the source code or to lift the binary to an *intermediate representation* in order to perform static analysis ([21]), or they make use of *symbolic execution*, associating symbolic expressions to addresses and trying to solve the constraints

they represent in order to compute the base address of a memory section ([23]). For what concerns the static analysis approach, we must consider 2 situations.

Source code required: closed-source software is always released without making the source code publicly available, thus preventing the analysis.

Binary lifted to IR: static analysis performed on lifted binaries may generate many **false positives** or **false negatives** due to the lack of semantic information about symbols and variables which are lost during compilation.

The symbolic execution approach, instead, can be applied to binaries directly. However, symbolic execution is known to be subjected to *path explosion*, which may slow down or even prevent analysis with higher program complexity.
Moreover, in order to avoid execution getting stuck, symbolic execution engines usually use *models* of the most frequent and complex library functions. Therefore, if the model oversimplifies the actual function, the symbolic execution may generate many **false positives** or **false negatives**, and it can still get the analysis stuck if it is not implemented at all.

We developed *MemTrace*, a new tool which makes use of *Dynamic Binary Instrumentation* (DBI) to detect **uninitialized reads** in a binary and report the memory overlaps that are generated during its execution.
We then paired our tool with a well-known fuzzer (*AFL++*) in order to try and explore as many execution paths as possible and report all the overlaps that may happen during binary's execution.
Finally, since it is even possible that a binary changes its behavior according to the arguments it is executed with, we leveraged the fuzzer in order to try and perform **command-line arguments fuzzing** as well.
We tested our tool with a set of binaries having known vulnerabilities. More specifically we tested the tool with 4 *real-world* binaries; 3 binaries from *Capture The Flag* (CTF) competitions; and 5 binaries from *Cyber Grand Challenge* (CGC) [3].
In most cases, the tool, paired with the fuzzer, was able to automatically report the known vulnerability.
In all the cases, however, the tool was able to report the vulnerability, when it was triggered by a manually crafted input.

In summary, our main contributions are the following:

- We leveraged DBI to perform a new kind of dynamic binary analysis, which aims at reporting memory overlaps, i.e., uninitialized reads together with write accesses

overlapping the same memory location

- We used a fuzzer to explore a program's *Control Flow Graph* (CFG) and increase coverage
- We leveraged the fuzzer to perform also command-line arguments fuzzing

1 | Motivation

Mitigation techniques implemented by operating systems and compilers make it more difficult to write exploits that allow to hijack control flow and, therefore, execute arbitrary code. The main and most known techniques are *stack canaries*, $W \oplus X$ and *ASLR*.

Stack canary

A *stack canary* [29] is a mitigation that is usually implemented as a cooperation among the kernel, the compiler and the runtime libraries and that is useful to try and detect return address tampering attempts performed through the exploitation of *buffer overflows*. During compilation, the compiler will add some code to the program in order to be able to (1) store the canary between the return address and local variables during functions prologues and (2) check integrity of the canary during functions epilogues. Then, when the program starts (i.e., during a call to *execve* system call), the kernel writes a random value in the memory of the new process and the runtime library functions will use this random value to compute the actual canary and store it in a well-known memory location. The compiler generated, for each function prologue, the instructions required to load the global canary from memory and store it on the stack.

The actual stack frame layout may be **platform-dependent**. Nevertheless, the usual **general** layout expects the return address to be pushed on the stack before the local variables, as depicted in Figure 1.1. In order to be effective, the *stack canary* must be stored between the stored *return address*, which is the address of the instruction executed right after the current function returns, and the current function's local variables. This way, if an attacker tries to exploit a buffer overflow to overwrite the saved return address, it will also overwrite the stack canary. Simply overwriting the canary is not a problem, and the function will continue executing as normal. However, during function's epilogue, right before it returns to its caller, the compiler added the code required to check the value of the canary. If the value stored in the function's frame is not the same as the one stored in memory during program's startup, the program will *abort*, avoiding executing the *return* statement and therefore not jumping to whatever has been written in the

return address memory location, thus not executing attacker's code.

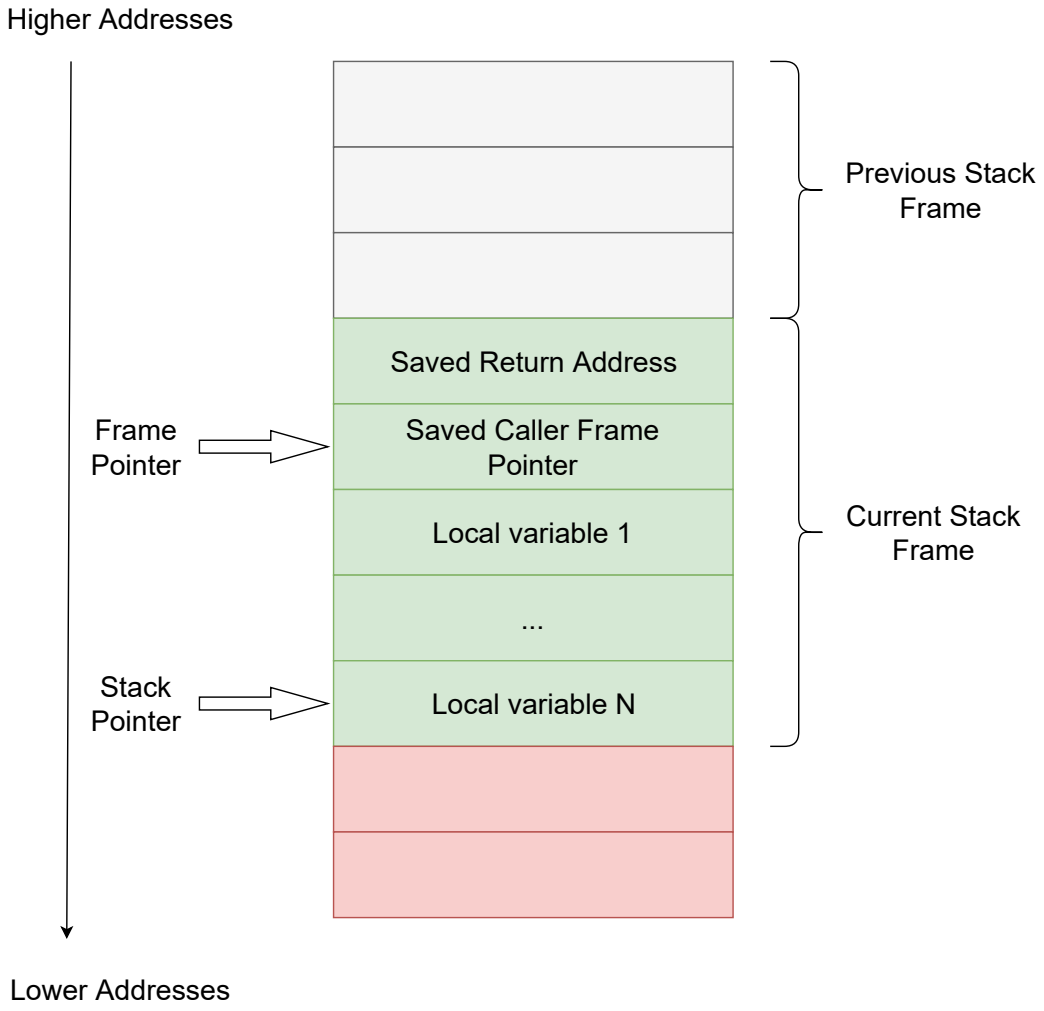


Figure 1.1: Stack frame memory layout

W ⊕ X

It stands for *Write XOR Execute*. It is a mitigation technique implemented by operating systems which allows a certain virtual memory page to be **either** *writable* **or** *executable*, never both. By doing this, an attacker cannot be able to write **arbitrary code** in memory (e.g., on the stack) and then execute it by replacing the stored *return address* of the current function with the address of the written memory location. Let us consider the two possible cases:

Writable, not Executable: the attacker can write arbitrary code at any address $\langle Addr \rangle$ in the range of the memory page. Assuming it can also replace the saved *return*

address with $\langle Addr \rangle$, an **error** is thrown as soon as the *return* statement is executed, as the page the address belongs to will be recognized as **not executable**.

Executable, not Writable: an **error** is thrown as soon as the attacker tries to write anything inside any address belonging to the memory page, thus aborting execution.

Address Space Layout Randomization

ASLR is implemented by the operating system which sets the base of each memory section (e.g., *stack, heap, code, libraries, etc*) to different addresses on each new execution of a program. This prevents an attacker to be able to execute a program a first time to retrieve interesting addresses (e.g., stack/heap buffers, canaries, functions...) and use the acquired knowledge on a second execution of the same program. Let us consider the simple example of Listing 1 and assume it has been compiled with neither *stack canary* nor $W \oplus X$ enabled.

```
1  #include <stdio.h>
2
3  void func(){
4      char buf[256];
5
6      scanf("%s", buf);
7  }
8
9  int main() {
10     func();
11     return 0;
12 }
```

Listing 1: Simple buffer overflow example

Function *main* will call function *func*, that simply declares the static *char* buffer and fills it by calling *scanf*. *scanf* does not accept a length parameter, and the passed format string does not specify a maximum length. This means that it is possible to completely fill the buffer and write beyond that, and therefore, it is possible to overwrite the value of the saved *return address*. If addresses were the same in 2 subsequent executions of the program, the attacker could easily write an exploit in 2 phases:

1. Execute the program inside a debugger and identify the offset of the saved *return address* from the end of the static buffer *buf* and the address of a **writable** and **executable** buffer (since $W \oplus X$ is **disabled**, it can be *buf* itself)

2. Execute the program inserting $\langle buf\_len \rangle + \langle ret\_addr\_offset \rangle$ random bytes fol-

lowed by the bytes composing $\langle buf\_addr \rangle$, where $\langle buf\_len \rangle$ is the length of buffer buf; $\langle ret\_addr\_offset \rangle$ is the offset of the memory word containing the *return address* from the last byte of buf; and $\langle buf\_addr \rangle$ is the address of the writable and executable buffer filled with **arbitrary code**.

With *ASLR* enabled this kind of two-steps approach will not work, because addresses will be different during the second execution.

Mitigation techniques bypass

Stack canary, $W \oplus X$ and *ASLR* have all a thing in common: they can be bypassed if we are able to **leak** information from program's execution. This simply means that we can still exploit a vulnerability even if mitigation techniques are enabled, but it requires having more information, making it more difficult. Of course, since their objectives are different, also the information required to bypass them will be different.

Stack canary: if we were able to leak the canary that is stored on the stack during a function's prologue, we could still exploit a *buffer overflow* vulnerability quite easily. Indeed, knowing the value of the canary, we can simply overwrite it with its own value, so that the canary check executed right before the *return* statement is passed successfully, thus not aborting program's execution. So function's execution proceeds normally, executing the *return* statement and therefore jumping to whatever address we stored instead of the actual *return address*.

W $\oplus$ X: we are not allowed to write arbitrary code in a buffer and then execute it because a memory page is either writable or executable. However, usually, there are a lot of functions in a single executable, especially if it makes use of external libraries, so we can try to take advantage of existing executable code. Even better, we don't really need to execute whole functions: we can jump to small sequences of instructions (gadgets) that allow us to move data to/from memory and jump to new sequences of instructions until we reach our goal (e.g., spawn a shell). In any case, we need to know the *target address*, i.e., the address of the instruction we want to jump to. This is quite easy if *ASLR* is disabled, as we can perform a 2-steps exploitation as described for the example in Listing 1. If instead *ASLR* is enabled (usually it is enabled **by default**), we need to leak an address that allows us to compute the *target address*.

ASLR: if enabled, it effectively randomizes the base address of each memory section allocated inside a process' address space. For clearness sake, and WLOG, consider

the *stack* as a specific memory section and assume we are interested in a stack address. Let us call $\langle Addr \rangle$ the target address and $\langle Base \rangle$ the address of the base of the stack. In subsequent executions of the same program $\langle Addr \rangle$ and $\langle Base \rangle$ values will change because of *ASLR*. However, $\langle Offset \rangle = \langle Base \rangle$ - $\langle Addr \rangle$ will be constant over all the executions. The same is true for **all** the memory sections allocated in the process' address space, including the ones belonging to external libraries. This means that it is enough to know just 1 address of a certain memory section to be able to compute **any** address of the same section. So, if we are able to leak any address belonging to the same section the target address belongs to, we will be able to break *ASLR* and compute our target address.

It is worth noting that usually these mitigation techniques are all enabled **by default**. So, if we want to write a working exploit, we will need to bypass all of them.

While it is not the only way that allows to obtain a leak, one possibility is to take advantage of **uninitialized reads**. Indeed, since memory is a very limited resource, programs always reuse the same memory locations multiple times. This is particularly true for the stack and the heap, which are the memory sections where most of the variables of a program are stored.

The stack is used to allocate function frames and the so-called *automatic* variables, which are automatically allocated and deallocated when program's flow respectively enters or exits variables' scope. *Local* variables are automatic variables whose scope is the function that declared them. This means that they are automatically allocated on the stack when the function begins (during function's prologue) and deallocated when it returns (during its epilogue). This continuous allocation and deallocation of memory allow local variables of a function to use again the same space that was previously occupied by the stack frame of another terminated function, thus creating unintended **overlaps** in memory. As a consequence, if the program uses or prints the value of an uninitialized variable, it will perform an uninitialized read that may allow us to retrieve the value of whatever was written in the memory location now occupied by the uninitialized variable itself (e.g., another variable, a canary, an address, etc). As an example, consider the program in Listing 2.

func1 is executed first, which declares 2 local variables, respectively of types *int* and *char** and initializes them before returning. Then *func2* is called, which still declares local variables of type *int* and *char**, but does not initialize them. *func1* is finished, so *func2*'s frame will occupy the same memory which was occupied by *func1*'s frame and, since it also declares the same number and types of variables, *m* and *ptr* will still contain

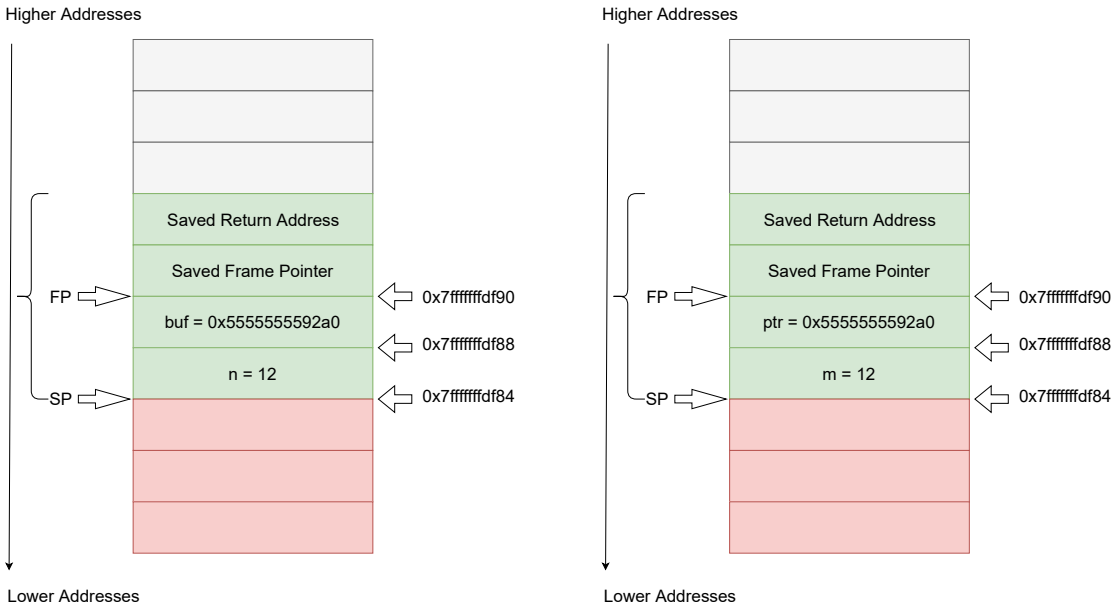
```c
#include <stdio.h>
#include <stdlib.h>

void func1(){
    int n;
    char* buf;

    n = 12;
    buf = (char*) malloc(sizeof(char) * 10);

    printf("n=%d\nbuf=%p\n", n, buf);
}

void func2(){
    int m;
    char* ptr;

    printf("m=%d\nptr=%p\n", m, ptr);
}

int main(){
    func1();
    func2();
    return 0;
}
```

Listing 2: Stack overlap example

the bytes related to variables *n* and *buf* of the previously executed *func1*. Figures 1.2a and 1.2b show the layouts of stack frames right before the call to *printf* for both *func1* and *func2*, respectively, also showing the addresses and the values of the local variables.

The heap, on the other hand, is a memory section where *dynamically allocated* objects are stored. This is usually handled manually by the programmer, which uses library functions to allocate and deallocate chunks of memory on demand. Even in this case, however, chunks are deallocated when the object they store is not useful anymore and they can be allocated again whenever a new compatible chunk is requested. Although the situation and the modality for allocations and deallocations are different, what we said for the stack is valid also for the heap. Indeed, when a chunk is deallocated and reallocated again, it will still contain the bytes related to the old, deallocated object. So, once again, some unintended **overlaps** may be generated that allow to exploit an uninitialized read to obtain a leak.

Nevertheless, it is not easy to detect uninitialized reads and, even if we were able to find one, it is often very difficult to understand from where the read bytes come from and, therefore, understand if and how I can obtain a useful leak. The most usual way of



(a) Frame layout for *func1* (b) Frame layout for *func2*

Figure 1.2: Overlapping stack frames example

approaching a program looking for some leaks is by manually performing a combination of **static** and **dynamic** analysis using tools like *decompilers* and *debuggers*. But such kind of analysis may require a large amount of time. Even with simple programs, it may require **hours** to analyze the possible memory overlaps to try and obtain a leak; while the analysis of more complex programs may also require **days** or **weeks** of analysis.

1.1. State of the art

There are some existing tools that might help looking for leaks.

Memcheck

Memcheck [36, 42] is a dynamic analysis tool implemented using *Valgrind* [32] as a *Dynamic Binary Instrumentation* framework. It is designed as a **memory error detector**, and can, therefore, detect and report the most common problems with memory management. As stated by [42], the most common memory errors that *Memcheck* is able to detect are:

- "illegal" accesses[1] (e.g., heap/stack overrun, use-after-free)

[1]These are memory accesses that are perfectly legal from a permission point of view, but that access

- usage of **uninitialized values**
- incorrect heap management
- overlapping *src* and *dst* pointers in *memcpy* and similar
- memory leaks[2]

Among the problems that *Memcheck* can detect, there are the usages of uninitialized values. The tool will therefore report the address of instructions that use uninitialized values, thus helping perform the first step toward the discovery of possible leaks. This, however, is not the actual objective of the tool. Indeed, *Memcheck* is designed to be able to report a large variety of memory errors that can reside in a program. This is mainly done to simply make the developers aware of the possible error, which can cause a crash in the program, or be an exploitable vulnerability. The real objective of the tool is, therefore, helping find memory errors to support debugging and allow the developers to easily track down and correct them. Moreover, being a **dynamic analysis** tool, it will return as a result a report containing the possible memory errors of a single execution path of the program. This means that, if nothing is reported, it does not mean that the program does not contain any memory error, but that the execution of the program with the given parameters does not contain memory errors. For instance, consider Listing 3. Function *func* accepts an *int* parameter and, according to its value, will execute a branch of the *if-statement* or the other. The *then* branch will perform an uninitialized read. However, if *Memcheck* is launched with an input that never triggers the *then* branch, it will not report any error.

Another main drawback of *Memcheck* is the number of reported errors. Indeed, since it is thought to support software correction, it will report **all** the instructions that may be involved in any of the memory errors *Memcheck* can detect. Even for the small example in Listing 3, *Memcheck* reports, according to the taken paths[3], up to 20 errors due to the single uninitialized read at line 8. If the program is compiled keeping debugging information (e.g., using flag *-g* with *gcc*) the tool is also able to print which line and function caused the error, thus allowing a developer to easily track it down and fix it. After an error is fixed, it is advisable to re-execute the analysis, because more than one reported error may have been related to the same cause. In our example, for instance, all the reported errors would be fixed by fixing the only instruction leading to an uninitialized

portions of memory that should not be accessed, because not considered in use

[2]Here it is intended as allocated memory whose pointer has been lost, and that therefore cannot be freed anymore

[3]Based on the sequence of numbers passed to the function, *Memcheck* reports a different number of errors. 20 is the maximum value achieved in just a few tests

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

void func(int n){
    char* buf;
    if((n & 1) == 0){
        printf("%s\n", buf);
    }
    else{
        buf = strdup("Initialized string");
        printf("%s\n", buf);
        free(buf);
    }
}

int main(){
    int n;

    scanf("%d", &n);
    while (n != 0){
        func(n);
        scanf("%d", &n);
    }

    return 0;
}
```

Listing 3: Conditional uninitialized read

read (i.e., the call to *printf* in the *then* branch).

The high number of errors reported by *Memcheck* could make it difficult to find an uninitialized read that may lead to a possible leak in more complex programs. Moreover, even if we were able to find an interesting uninitialized read, no information is provided by *Memcheck* about what can be read from that or where the read bytes come from. So, it would still require the user to perform a manual analysis in order to understand if it can obtain a leak.

Sleak

Sleak [23] makes use of a combination of *static analysis* and *symbolic execution* in order to try and find paths in the binary that lead to possible **leaks**. As a first step, *Sleak* lifts the binary to an *intermediate representation* that allows to build the CFG more easily. The CFG is then used to perform a set of static analysis whose objectives are (1) identify *address variables*, i.e., variables that might contain an address; (2) identify

output functions, i.e., functions that print at least 1 of their parameters; and (3) identify **potentially** leaking paths, i.e., paths that might print an address through any of the output functions. The information retrieved from the static analysis is then used to perform the *symbolic execution* of the identified paths.

Symbolic execution is an analysis technique that allows to set a symbolic expression as a value for inputs or variables and to execute the instructions of the program in order to update the symbolic expression, which simply represents a set of constraints over the bits of the symbolic value. In this case, *Sleak* associates a symbolic expression to each address variable identified during the static analysis and starts the symbolic execution. The main problem of symbolic execution is **path explosion**. Indeed, if during execution the condition of a branch depends on a symbolic expression (even partially), the symbolic execution engine cannot decide which branch to take. So, it simply takes both of them, by splitting execution into 2 independent states and adding to the symbolic expression the constraints required to execute that specific branch. Given the huge number of branches in a single program, it is possible that the execution state splits so many times that it is infeasible to actually terminate the analysis due to the time required to carry on each execution state. In an effort to deal with path explosion, *Sleak* does not execute the whole program in the symbolic execution engine, but it takes advantage of the information about possible leaking paths gathered through the static analysis in order to symbolically execute only those interesting paths. In order to initialize the state of the program to perform a coherent symbolic execution of a certain path, *Sleak* combines it with a real execution of the program itself, which allows to extract context information and use it as a starting point for the symbolic execution.

Once the symbolic execution of the paths is terminated, *Sleak* collects the set of final symbolic expressions and computes the address of interesting objects (e.g., base addresses of the stack, the heap or other memory sections) by simply providing to the constraint solver a real value of the output of the program. Note that this kind of approach is actually able to detect **all** types of possible leaks, not only the ones due to *uninitialized reads*. However, although *Sleak* tries to deal with path explosion, it may still be a problem, because some of the leaking paths identified with static analysis may still contain many branches. Moreover, performing static analysis on a lifted binary may be a complex task, because recovering a precise CFG is not straightforward. Indeed, as explained by [18], it may require dealing, for instance, with *indirect control flow transfer* instructions (e.g., indirect calls) or optimizations that cause an unusual layout or usage of instructions, like functions with multiple entry points, code shared among multiple functions or *call* instructions used to push the *instruction pointer* on the stack. Since the quality of results

of data flow analysis also depends on the quality of the CFG, any inaccuracy of the CFG may affect its results, thus possibly detecting many potentially leaking paths and therefore increasing the execution time of the whole analysis.

Finally, *Sleak* needs to detect output functions to find interesting paths for symbolic execution. If the program has been compiled stripping all the symbols information, the tool needs to find by itself the boundaries of functions in the executable and identify which of them are actually output functions. *Functions boundaries detection* in stripped binaries is still an open research topic as it needs to address many challenges ([9, 28]). Since it is not its main concern, *Sleak* performs boundaries detection by simply scanning the whole binary looking for functions prologues ad epilogues. After boundaries detection is done, *Sleak* uses a heuristic to distinguish output functions: a function is marked as an *output function* if it invokes the *write* system call **and** it passes one of its parameters to the invoked *write* system call. While this approach should work well with the most simple programs, it might not work with more complex cases like real-world stripped binaries, where functions prologues and epilogues can be optimized or removed and function calls can be avoided by means of procedure inlining.

1.2. Goals

Our objective is to design a tool that allows analyzing any binary executable looking for **memory overlaps** that may lead to information disclosure and that circumvents or addresses the main limitations of the existing similar tools. With *memory overlap*, we intend a set of instructions composed by an **uninitialized memory read** and **all** the memory writes that have previously written in the same *memory location* and have not been completely overwritten, where a memory location is identified by an *address* and a *size*.

Since *symbolic execution* has some very strict intrinsic limitations, we want to simply avoid that. Also, since we are going to deal with memory accesses, *static analysis* may be a quite complex approach due to **memory aliasing**, which would force us to perform *alias analysis*. Besides not being always possible, it might be quite inaccurate, especially since we would work with a CFG built from a lifted binary. Therefore, we decided to leverage **Binary Instrumentation**.

Binary instrumentation allows to add, remove or modify the control flow of a binary executable in order to modify or observe its behavior. There are, mainly, 2 distinct approaches to instrumentation:

Static Instrumentation: instrumentation code is actually added inside the executable. While it should be faster than dynamic instrumentation, its main drawback is the lack of run-time information (e.g., memory addresses), which might make it difficult to implement analysis functions. Also, adding or modifying instructions in a binary executable might break the program itself (e.g., if the inserted code modifies a register which is later required). For this reason, it is usually done when the source code is available, so that it is possible to perform the instrumentation at compile time, when the compiler has all the information to add instructions without compromising program's behavior.

Dynamic Instrumentation: instrumentation code is called while the binary is executed, so analysis functions will have all the run-time information available. This makes it easier to implement analysis functions, but introduces a bigger overhead due to the number of jumps from application code to analysis code and vice versa.

Again, we want to analyze memory accesses of a binary executable, so *Dynamic Binary Instrumentation* is the approach that better fits our needs.

2 | MemTrace

2.1. Overview

As shown by Figure 2.1, *MemTrace* takes an executable as input together with a set of command-line arguments and some user input, if required by the executable itself. The main idea behind *MemTrace* is to execute the binary with the given arguments and keep track of all the executed memory accesses. This way, as soon as it detects an uninitialized memory read, *MemTrace* can report it and look backward at which memory writes were executed last on the same memory location.

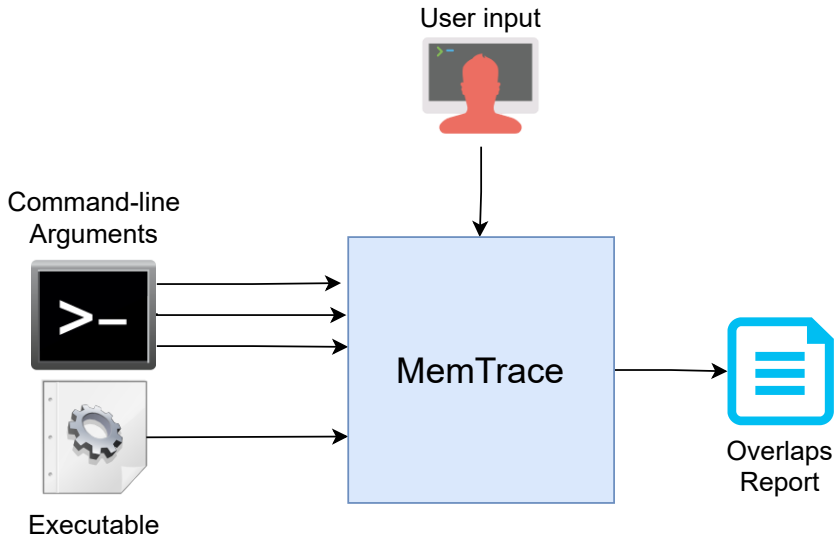


Figure 2.1: *MemTrace* inputs and output

As a means to keep track of the *state* of bytes in memory (i.e., either **initialized** or **uninitialized**), *MemTrace* uses a **shadow memory** which associates a bit to each byte of memory used by the analyzed application, so that every time it detects a read memory access, it can easily and quickly check the state of the read memory location.

During a program's execution, it is not rare that some values are copied into other registers or other memory locations. So, in order to deal with these *data transfers*, we also designed

a **taint analysis** that allows *MemTrace* to keep track of copies of *uninitialized bytes* and therefore detect also *indirect* uninitialized reads and usages of uninitialized bytes.

2.2. Approach

MemTrace makes use of DBI in order to analyze the instructions executed by a binary and detect the ones performing a *memory access*. Memory accesses are divided into 2 categories: *write accesses* and *read accesses*. Whenever a memory access is detected, *MemTrace* will store information about it, so that it will be possible to review the whole history of executed memory accesses. By doing this, *MemTrace* is able to group *read memory accesses* with all the *write memory accesses* that overlap the same memory location and are not completely overwritten before the execution of the read access itself. In order to achieve its goal, however, *MemTrace* must also be able to recognize read accesses that read uninitialized data. In order to do that, *MemTrace* must know, at any moment, the state of memory bytes. For this purpose, we designed and implemented a **shadow memory**.

As the name suggests, a *shadow memory* is a data structure that mirrors the actual memory used by the application, thus allowing to store information about memory content. In this case, *MemTrace* leverages the shadow memory to store state information about every single byte of memory, which can be either *initialized* or *uninitialized*. So, when *MemTrace* detects a read memory access, it can query the shadow memory to check whether the data the program is going to read is initialized or not.

Since our objective is to try and report *uninitialized reads* that possibly lead to a leak, not all the uninitialized reads performed during a program's execution are really interesting. Indeed, the bytes read by an uninitialized read access may be simply loaded into a register, but then they may never be used by any instruction, or they may be used only by a *cmp* or *test* instruction to evaluate the condition of a branch. In those cases, the uninitialized read cannot lead to a leak. Moreover, it often happens that the same value is copied in more registers or even in other memory locations. In order to be able to keep track of usages, transfers and copies of uninitialized bytes, *MemTrace* implements a *taint analysis*. A *taint analysis* is a sort of *data-flow analysis* which simply marks some data as *tainted* according to some criterion and keeps track of those tainted data to mark as *tainted* also every other piece of data that depends on that. Since it requires keeping track of uninitialized bytes, the taint analysis performed by *MemTrace* marks as *tainted* all the bytes that are in the *uninitialized* state when a memory read accesses them. Then, the taint analysis proceeds following the flow of the tainted bytes and marking as *tainted* all

of their copies. In summary, the taint analysis is helpful to (1) reduce **false positives** by ignoring uninitialized reads whose bytes are never used by any other instruction (and therefore cannot lead to a leak); (2) reduce **false negatives** by detecting usages of copies of uninitialized bytes; and (3) detect and correctly report **indirect uninitialized reads**.

With the term **false positives**, we mean overlaps where the *uninitialized read* access is actually executed, but it is not caused by errors in the program (e.g., it is caused by compiler optimizations) or the read uninitialized bytes are never actually used; instead, we refer as **false negatives** those uninitialized read accesses actually performed by the program whose bytes are used by at least an instruction but which are not detected by *MemTrace*. Finally, we call **indirect uninitialized read** any read access that reads copies of some uninitialized bytes which have been stored in another memory location (e.g., by a call to *memcpy*).

In order to achieve its objectives, the taint analysis required to design a new component. Indeed, while the shadow memory allows to keep track of loads and stores of uninitialized bytes, it does not allow to follow the propagation of uninitialized bytes within registers. For this reason, we designed a **Shadow Register File**, that, as the name suggests, works similarly to the shadow memory and is responsible for keeping information about the bytes stored inside registers, so that *MemTrace* can know, in every moment, their state.

By using both the shadow memory and the shadow register file, the taint analysis is always able to detect and manage usages and copies of uninitialized bytes, thus successfully reducing the number of false positives and false negatives. However, being able of following the flow of uninitialized bytes is still not enough to allow *MemTrace* correctly report the indirect uninitialized reads. Indeed, thanks to the taint analysis, *MemTrace* is certainly able to detect when the program is reading from a memory location where a copy of some uninitialized bytes have been stored, but it will not be able to trace the original memory read access that first loaded the uninitialized bytes from memory.

Let us consider, for example, Listing 4. The call to *printf* at line 19 will read bytes copied by the previous call to *strcpy* at line 16. Those bytes were not initialized after the allocation at line 15, therefore, the call to *printf* will perform an **indirect uninitialized read**. However, the bytes read and used by *printf* are actually initially loaded from memory by *strcpy*, which stores a copy of them inside *stackBuf*. In such a case, we would like *MemTrace* to report the instruction I_{read} within *strcpy* that performed the original uninitialized read. To enable this capability, besides propagating the state of the copied bytes, the taint analysis must also propagate the **origin** of the uninitialized bytes. In the context of taint analysis, we call an *origin* the first memory access that loaded the

uninitialized bytes stored in a register or in a memory location. So, in the example of Listing 4, the origin of the uninitialized bytes read by *printf* is instruction I_{read} executed by *strcpy*.

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

void func(){
  char* heapBuf;
  char stackBuf[64];

  // Write something in the heap buffer
  heapBuf = (char*) malloc(sizeof(char) * 64);
  strcpy(heapBuf, "Lorem ipsum dolor sit amet, consectetur adipiscing elit vivamus");
  free(heapBuf);

  // Using glibc implementation, the following malloc will return the same memory location occupied by the previous allocation of heapBuf
  heapBuf = (char*) malloc(sizeof(char) * 64);
  strcpy(stackBuf, heapBuf);

  // +16 becauce glibc implementation overwrites the first 16 bytes when a chunk is allocated again, thus initializing them
  printf("%s\n", stackBuf + 16);
}

int main() {
    func();
    return 0;
}
```

Listing 4: Indirect uninitialized read example

In order to efficiently propagate origins, *MemTrace* makes use of a **Tag Manager** which assigns an integer *tag* to each origin, so that it is sufficient to propagate only the tag, instead of propagating all the information about the origin itself. Given an integer *tag*, the tag manager is of course able to return a reference to the corresponding memory access, so that it is possible, at any moment, to retrieve information about the origin of uninitialized bytes.

While *MemTrace* can be used as a standalone tool to detect the *uninitialized reads* performed by a certain execution path of a program, it is actually designed to be used in combination with a fuzzer. So, the analysis should be as fast as possible. For this reason, the report generated by *MemTrace* is not in a textual form, but it is in a custom binary format. This is simply done to avoid spending time formatting in a human-readable way

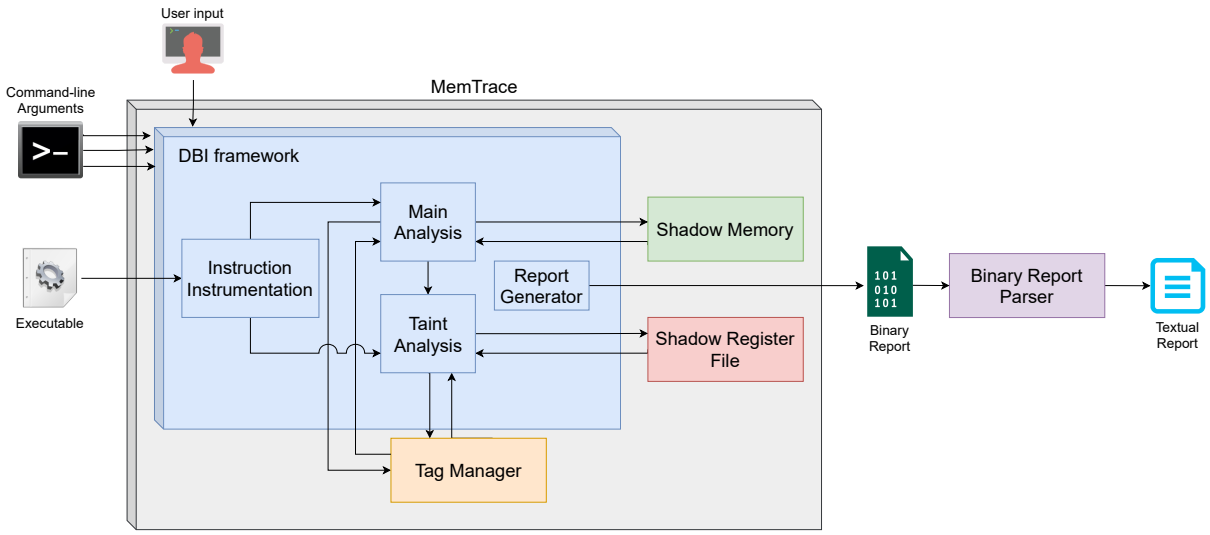


Figure 2.2: Block diagram of *MemTrace*

all the information about the detected overlaps, thus reducing the time required to generate the report file. In order to generate the textual, human-readable report, it is therefore required to use an external parser, which is able to parse the binary report and print the contained information in a set of well-formatted tables.

Figure 2.2 shows the structure of *MemTrace* and the interactions among its main components.

2.3. Implementation

MemTrace has been implemented as a dynamic binary analysis tool using *Intel PIN* [30] as the underlying DBI framework. It is composed of a set of *analysis functions*, each of which handles a different situation, so that their implementation is as simple as possible. Having more independent analysis functions also allows to simply avoid executing them when they are not needed, instead of introducing a branch for each different case in a single huge function.

Each executed instruction is analyzed and managed accordingly to its type. Instructions are indeed broadly distinguished into 2 types: if the instruction performs at least a memory access (either *write* or *read* or both), the main analysis function is executed in order to keep track of the performed access; while if it does not perform any memory access, another analysis function is executed, which will be better explained later in this section.

As mentioned in Section 2.2, a shadow memory is used in order to keep track of the state of memory bytes, which can be either **initialized** or **uninitialized**. Having just 2 possible

values, the state of a byte of memory can be represented using only 1 bit: as shown by Figure 2.3, it will be set to 1 if the corresponding byte is *initialized*; it is set to 0 otherwise. So, the shadow memory will associate 1 bit to each byte of memory used by the analyzed application, thus reducing by $\frac{1}{8}$ the amount of memory required to mirror its state. The shadow memory essentially works as an ideal *hash table*, thus uniquely associating each bit of the shadow memory to only one byte of the actual process memory, therefore avoiding collisions. Working as an hash table, *update* and *lookup* operations are very fast (i.e., *O(1)*), and the absence of collisions allows to avoid the linear cost due to the presence of multiple elements in the same *bucket*, thus making it very efficient. Even the "hash" computation, which we call *shadow address*, is very fast, as it simply requires to compute the *byte offset* of the accessed byte from the beginning of the memory and add the same value as a *bit offset* from the beginning of the shadow memory.

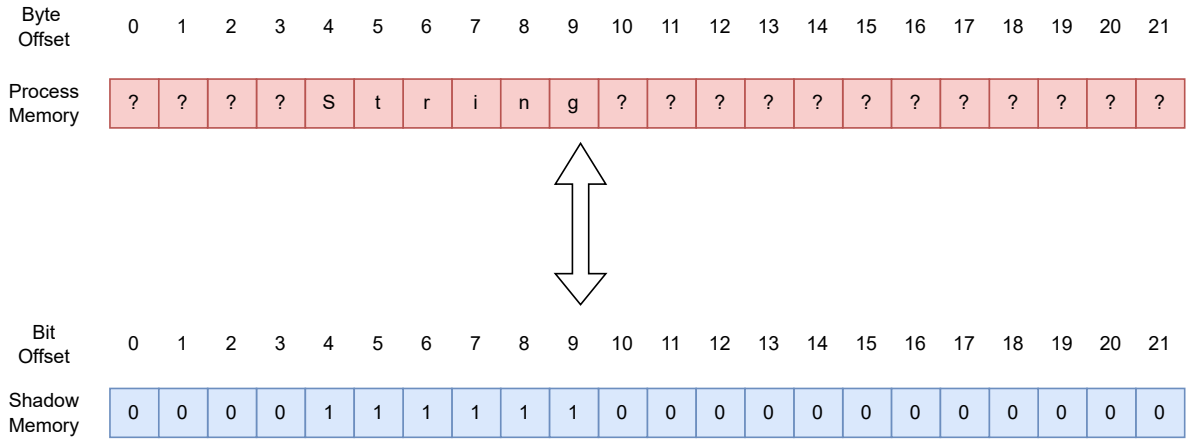


Figure 2.3: Shadow memory used by *MemTrace*

Unlike [31], *MemTrace* does not need to cover the whole process address space. Indeed, *MemTrace* is meant to keep track of **uninitialized reads** happening on the stack or on the heap, so it is sufficient to mirror only those memory regions with the shadow memory. Still, there are some difficulties that need to be addressed. First of all, both the stack and the heap may allocate memory pages that are not actually used by the process. This is done simply because the allocation of new memory pages is performed through the invocation of system calls, and is therefore considered an expensive operation. So, in order to avoid allocating memory pages too frequently, a whole block of memory pages are initially allocated for the stack and the heap. If some of these pages are never used by the program, it is not useful to mirror them with the shadow memory. Then, notice that the stack and the heap may require to allocate new pages during program's execution, so, *MemTrace* cannot know on process startup how many pages they will need. Finally, the program may allocate more than a single heap. Therefore, to keep track of the memory

accesses performed during execution, *MemTrace* must be able to handle all of them. For these reasons, *MemTrace*'s shadow memory is not implemented as a single huge block of sequential shadow addresses, but it is partitioned and allocated on demand.

In practice, the stack and each allocated heap will be mirrored by their own independent shadow memories, which will be composed of only a few pages at the beginning. If required, then, new pages will be allocated to a shadow memory, and, as depicted by Figure 2.4, the new page will be made *virtually sequential* with the previously allocated ones by adding them to a vector containing all the shadow pages belonging to the same region. This way, we can keep track of the state of each byte of process memory efficiently without consuming too much memory uselessly. Of course, the computation of the shadow address is made a bit more complex, as it is not sufficient anymore to compute the byte offset of the actual address and add it as a bit offset on the shadow memory. However, it is still very fast to be computed, as it only involves some arithmetic operations using division and modulo operators, just like an actual hash computation.

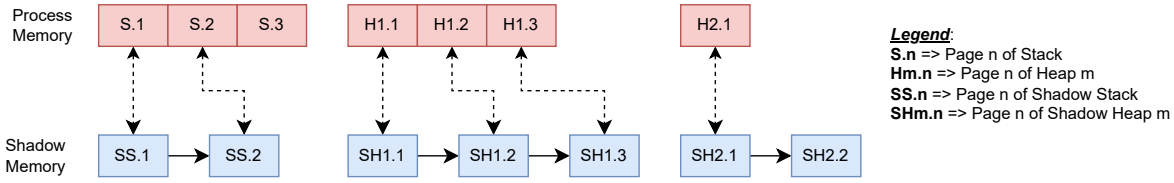


Figure 2.4: Shadow Memory Implementation

To deal with *indirect uninitialized reads*, *false positives* and *false negatives*, we implemented a *taint analysis* as a set of analysis functions and auxiliary data structures. In order to reduce the number of reported false positives, *MemTrace* should ignore those uninitialized reads whose bytes are not used by any other instruction, or which are used only by data transfer instructions (e.g., mov) or *cmp* instructions. Indeed, uninitialized reads usually simply load some values from memory to registers. In order to detect bytes usage, we need to check which registers are used as a source for the execution of an instruction. This is not a straightforward task, and in order to accomplish that, we needed to implement a Shadow Register File. Its objective is actually the same as the shadow memory, as it is meant to mirror the current state of each byte of every register in the processor. However, it is not implemented as the shadow memory due to some major differences. First of all, registers have a fixed size, so it is not needed to implement the *on demand* allocation of shadow pages, as we can allocate all the space we need on program's startup. Since the information stored in the shadow registers is the same stored in the shadow memory (i.e., byte's state is either initialized or uninitialized), we still associate a single bit of the shadow register to a whole byte of the corresponding architectural

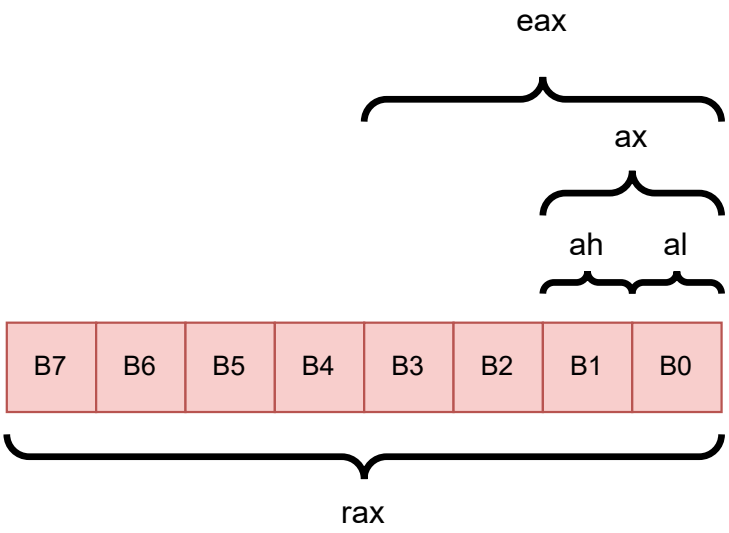


Figure 2.5: Aliasing set for register RAX

register, so one memory page is enough to allocate space for all the shadow registers of an x86_64 architecture.

While the on demand allocation is not requested, the implementation of the shadow register file is actually more complex than the shadow memory. This is because the physical register file of a processor contains many registers which can be very different both in size and in behavior. Indeed, modern processors usually make use of different sets of registers according to the type of instruction that will use them (e.g., integer, floating point, vector registers). Moreover, there are also sets of **aliasing** registers, which are registers that might be different in size and behavior, but that partially overlap. For instance, consider registers *rax, eax, ax, al, ah* in x86_64 architectures. As shown by Figure 2.5, these registers simply consider different portions of the same set of bytes. This means, for example, that if we write bytes inside register *rax*, also its aliasing registers will be written. Sometimes, also the other way around is true. For instance if we write register *eax*, register *rax* will be completely overwritten, filling the most significant bytes with 0.

The shadow register file hides all the complexities related to different types of registers and aliasing sets, exposing a very simple interface to the other components, which can use it as an intermediary to request to update or query a certain register. As can be inferred from the class diagram depicted by Figure 2.6, the shadow register file will then select the actual implementation of the correct register, which takes into account all the properties of the register itself, such as the size, the aliasing registers and their behavior

(e.g., overwrites its *super-register* or not).

By using the shadow memory and the shadow register file, we managed to implement an effective *taint analysis* as a set of analysis functions to be executed either before or instead of the main analysis. Indeed, regardless an instruction performs a memory access or not, the following analysis functions are executed **in order** and **before the main analysis** is executed:

checkSourceRegisters: checks the registers used as a source by the instruction. Requests the *shadow register file* to query them all and, if any of them contains at least 1 uninitialized byte, it retrieves the *origin* of the byte and reports it. This function is skipped if the instruction is a data transfer or a *cmp* instruction.

checkDestRegisters: checks the registers used as a destination by the instruction. It reinitializes them as *completely initialized* as it will be written by the instruction. If the instruction also loads uninitialized bytes into the same register, the main analysis function will update its state later.

After *checkSourceRegisters* and *checkDestRegisters* analysis are executed, the instruction is inspected to verify whether it accesses memory. If it does, the main analysis is executed. If the main analysis detects an uninitialized read there are 2 cases to consider:

Instruction is a data transfer: in this case, the instruction simply loads the read value into some register. The state of the corresponding *shadow register* is updated, and the memory access is left pending in a dedicated data structure. If any subsequent instruction will use any of the uninitialized bytes it read, the memory access will be reported, otherwise it will simply be discarded.

Instruction immediately uses the value loaded from memory: the memory access is immediately reported. Since it is not left pending, there's no need to keep track of the uninitialized bytes, because their origin has already been reported.

If instead the instruction does not perform any memory access, the *register propagation function* is executed **instead of** the main analysis. This function is responsible to propagate the state of each byte from the source to the destination registers. This last job is particularly complex and will be further discussed in Section 2.4.

To complete the discussion about *taint analysis*, we must finally speak about how we keep track of the origins of uninitialized bytes. When an uninitialized read occurs and simply loads bytes into some register, it is not immediately reported, but it is temporarily stored in an auxiliary data structure and will be permanently stored only when a subsequent instruction uses any of its uninitialized bytes. The data structure containing all the

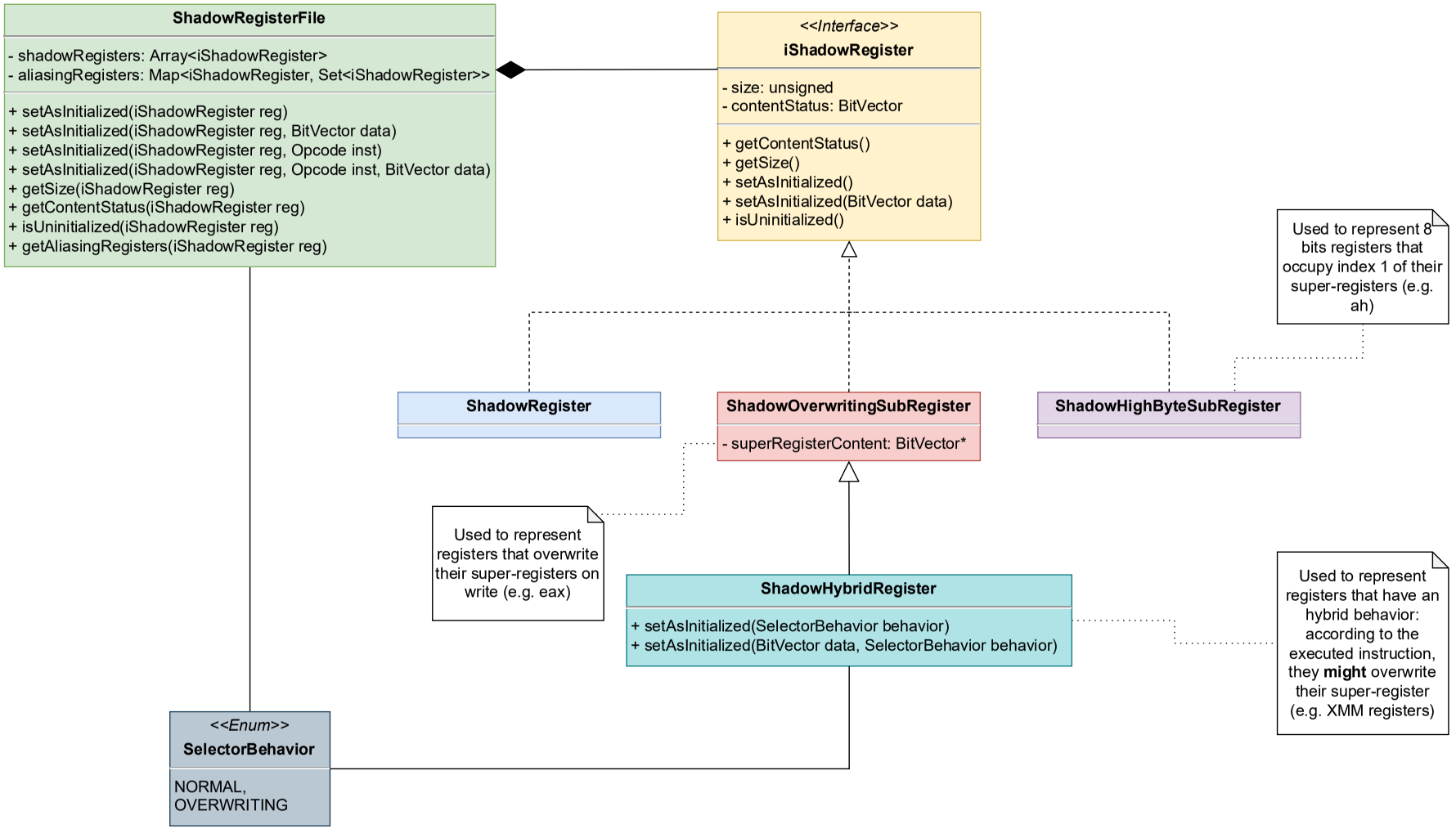


Figure 2.6: Conceptual class diagram of the ShadowRegisterFile

information about an uninitialized read is the *origin* for all the *uninitialized bytes* read by the corresponding instruction. In order to allow *MemTrace* to report the instruction that loaded the uninitialized bytes, it must propagate the information about their *origin*. The most straightforward method would be to copy the whole data structure representing the uninitialized read access every time the uninitialized bytes are propagated somewhere else. This, however, is not very efficient, as it requires performing many copies of a complex data structure composed of several fields. For this reason, we implemented a **tag manager**, which is responsible for uniquely associating an integer value, which we called **tag**, to a memory access.

The implementation of the tag manager is quite simple, as it mainly consists of a map associating a tag to a memory access. Since we also required a unique association, the tag manager also holds another similar map that does the opposite association, i.e., it associates a memory access to a tag (key and element of the map data structures are reversed). While this essentially doubles the space required to keep track of a single memory access, it also allows to quickly find out if a certain memory access already has a tag associated and, if it does, to retrieve the corresponding tag without scanning the whole tag/access map. Finally, the tag manager also holds a reference count for each tag. The reference count is not really necessary, but it allows to free memory allocated to the association of a certain tag when it is not useful anymore, thus reducing the total amount of space occupied by the maps. Indeed, whenever a reference count drops to 0, it means the associated memory access has been already managed or has been ignored for some reason (e.g., never used uninitialized bytes and the register holding its bytes has been overwritten) and therefore we can safely remove it from the maps. Without the reference count, the tag manager would still work correctly, but all the non-referenced memory accesses would be left in the maps, thus consuming memory.

2.4. Challenges

Memory accesses tracing

The first evident difficulty we had to face is related to the number of memory accesses performed in a program. Indeed, even if we write just a few lines of source code, after the compilation and linking phases the resulting executable may need to perform thousands or even hundreds of thousands of memory accesses. It would consume too much memory to keep track of them all and **it is not actually useful**. Moreover, we need to store information about the accesses we want to keep track of in some data structures and that is a non-instantaneous operation. In fact, every data structure has an asymptotic cost for

insertion, deletion and lookup operations. Therefore, if we insert such a high number of memory accesses in the same data structure, the time required to perform these operations may grow higher and higher, thus increasing the overall analysis execution time, possibly making the analysis infeasible with some complex binaries.

Since we are only interested in **uninitialized memory reads** and in the **memory writes** whose bytes are read by those uninitialized reads, we can discard all the other memory accesses. While this is quite easily done with read accesses, as we know whether the memory location it reads from is initialized or not when it is executed, it is more difficult for write accesses. Indeed, we cannot know or predict if a certain write access is writing bytes that will be later read by an uninitialized read, so we still need to keep track of every executed write access. However, in order to try and mitigate the problem, we make use of an additional *map* associating to each *AccessIndex*, which represents a memory location and therefore is composed by an **address** and a **size**, the last write access that wrote the memory location it represents. This way, every write access will overwrite the previous one that wrote something in the same memory location, instead of allocating more memory to add an entry in the data structure, thus discarding some of the write accesses that don't need to be tracked. Note that the additional map also allows to avoid checking the relative execution order of the write accesses performed on a certain memory location, as only the last one is actually stored. When *MemTrace* needs to store an uninitialized read access, it will also inspect this additional map, and will permanently store all the write accesses that overlap the uninitialized read, even partially.

By discarding the initialized memory reads and using the additional map for write accesses, we effectively reduced the analysis execution time and memory consumption, making the analysis feasible also for complex real-world binaries.

System call tracing

System calls allow any application to request a service to the operating system's kernel. Being defined by the system's ABI, system calls are **intrinsically** platform dependent. *Intel PIN* is able to detect when a system call is executed, but it cannot trace its behavior in a generic way because each system call has its own specific behavior. For this reason, it is not possible to simply rely on the DBI framework to get the memory accesses performed during system calls execution.

In order to solve this issue, we designed a **system call manager** whose job is to work as an intermediary between some analysis functions and a platform-specific **system call handlers** header file (Figure 2.8). To extract the memory accesses executed during a

certain system call, the following sequence of operations is executed:

1. *OnSyscallEntry* is executed by *Intel PIN* whenever a system call begins. It retrieves the system call number and its arguments and sends them to the *system call manager*.
2. *OnSyscallExit* is executed by *Intel PIN* right after a system call terminated its execution. It will send the return value to the *system call manager*
3. After the *system call manager* received the arguments and the return value of the executed system call, it will call a specific handler modeling the behavior of the system call itself, which will return the set of executed memory accesses, if any.
4. The *system call manager* simply returns the set of memory accesses to *OnSyscallExit*, which will add them to the set of memory accesses performed by the program.

The detailed sequence of requests and responses performed by each component of *MemTrace* can be seen in the sequence diagram depicted in Figure 2.7.

Using such an intermediary component allows to **decouple** the **generic** behavior of any system call from the **specific** behavior of each system call on a certain platform, thus allowing to easily extend the tool by replacing only the specific behavior implementation, if needed, keeping the intermediate *system call manager* and the analysis functions untouched.

Dynamic memory allocation functions

Objects that require to be allocated dynamically are usually stored on the *heap*. Unlike the *stack*, the heap usually requires to be handled manually by the developer, which can call specific functions in order to allocate or deallocate chunks of memory. These functions are usually implemented in a system library (e.g., *glibc*) and their behaviors are not standardized, making them intrinsically platform dependent. In particular, every implementation of the dynamic memory allocation functions may manage the heap in different ways (e.g., *brk* vs *mmap*) and may use a different layout for the metadata stored in a chunk. In order to be able and correctly keep track of memory accesses performed on dynamically allocated chunks, *MemTrace* requires to have some knowledge about the behavior of memory allocation functions.

Once again, we decoupled the **generic** behavior from the **specific** behavior of these functions in order to easily allow extending the tool. Indeed, as shown by Figure 2.8, the main analysis function will interact with an external header file requesting the information

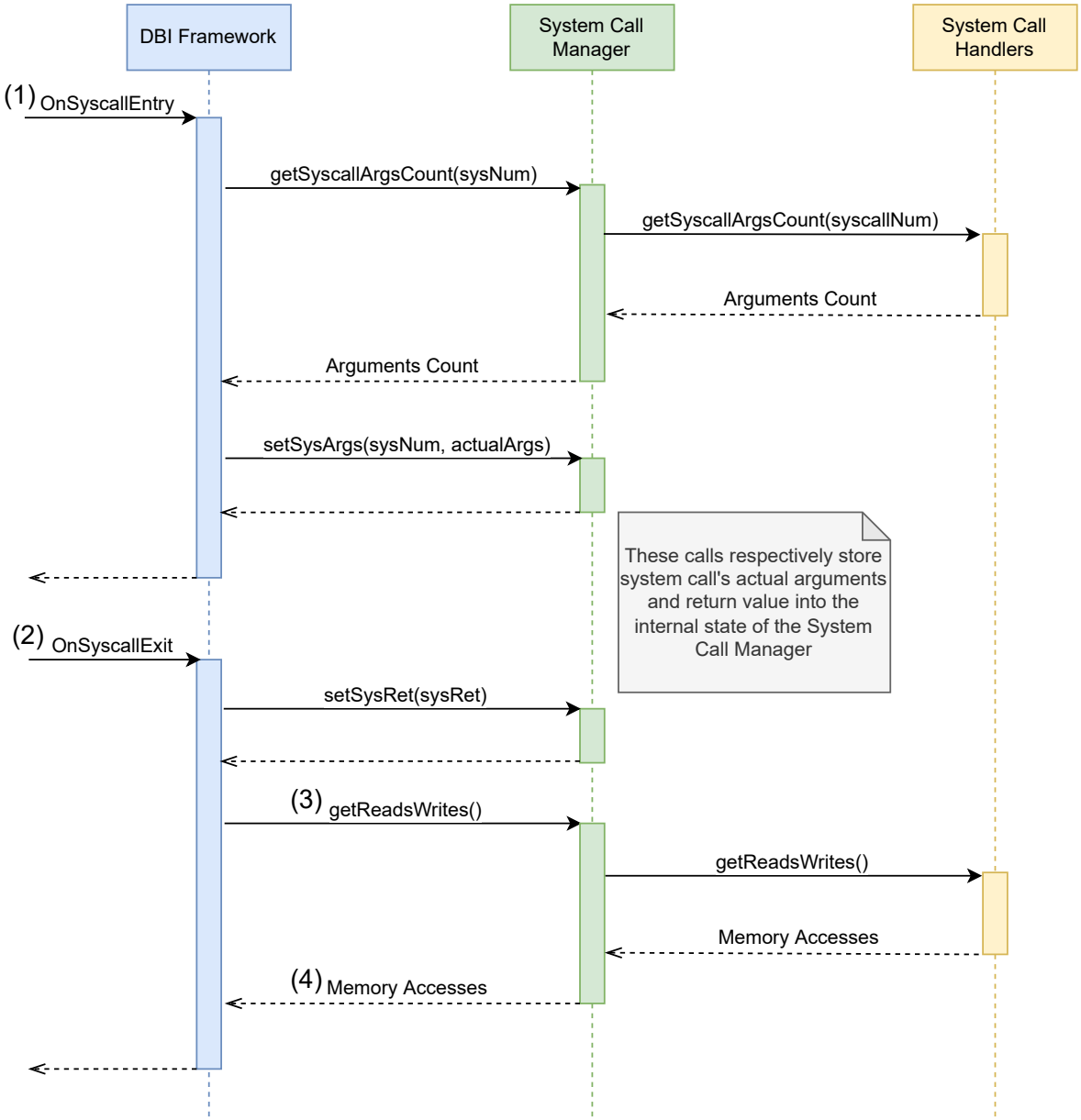


Figure 2.7: Sequence diagram of the System Call Manager

it needs by simply calling functions from a fixed interface whose objectives are summirized in Table 2.1. This way, the main analysis function can retrieve all the platform-specific information it needs and the tool can be easily extended to new platforms or even custom implementations of the memory allocation functions by simply implementing a new header file which adheres to the required interface.

Objective	**Reason**
Retrieve the beginning of the allocated chunk given the address returned by a call to *malloc* or similar	In some implementations, *malloc* returns an address different from the address of the allocated chunk to allow storing both metadata and the payload in the same chunk
Retrieve the size of the allocated chunk given the address of the chunk itself	*MemTrace* must know the size of the allocated chunk in order to correctly update the corresponding shadow memory
Retrieve the upper bound of the *main heap*	It is possible to allocate and use multiple heaps in the same program. For instance, on Ubuntu, using *glibc* implementation, the main heap is always increased by using *brk/sbrk* system call, while additional heaps can be allocated through a call to *mmap*. *MemTrace* supports the usage of multiple heaps, but if the *main heap* is managed through *brk*, it must compute the upper bound from the address of the *top chunk*[1], whose position w.r.t. the last allocated chunk may differ according to the specific implementation of *malloc*
Retrieve a set of pairs ($\langle Addr \rangle$, $\langle Size \rangle$) that should be considered uninitialized after a call to *free* (or similar)	The allocated chunks contain some metadata used by the memory allocation functions. On some systems, some of these metadata are written only once and kept untouched over all the allocations of the same chunk. So, if we consider those metadata as uninitialized after a call to *free*, we may generate many false positives as they may be read by subsequent calls to *malloc*. By implementing an ad-hoc function providing this kind of information, we can avoid those false positives, but of course, this requires in-depth knowledge about how *malloc* manages metadata

Table 2.1: Objectives of functions in the malloc handlers header file

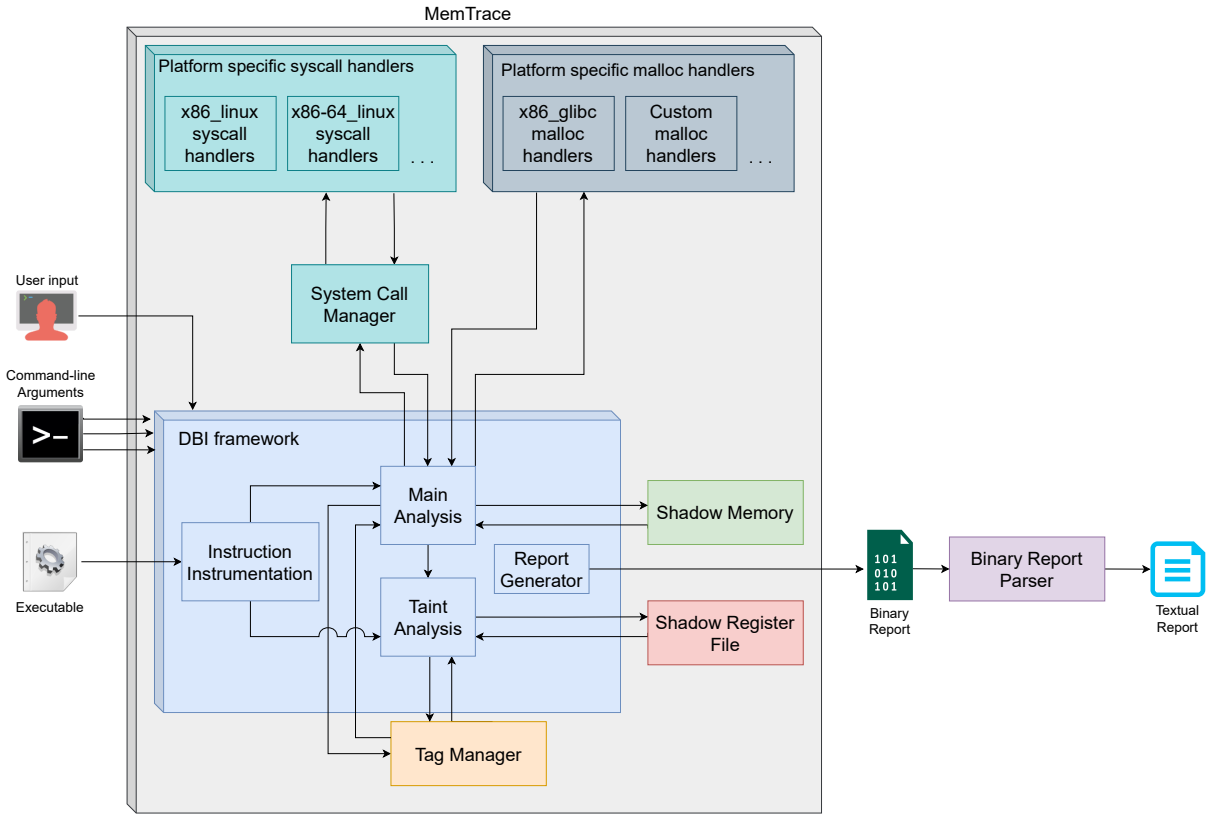


Figure 2.8: Complete structure of *MemTrace*

Heap shadow memory reinitialization

While it is clear that the effect of a call to *free* should reinitialize (i.e., set to an **uninitialized** state) the shadow memory associated to the freed chunk, it was not immediately clear **when** to reinitialize it. There are, indeed, 2 possibilities: reinitialize the shadow memory **before** the actual execution of function *free* or **after** its execution. Both the approaches have some drawbacks.

After free: *free* may write some metadata inside the chunk it deallocated that are later used by subsequent calls to *malloc*. By reinitializing the shadow memory associated to the chunk after the execution of *free*, subsequent calls to *malloc* will see the location of the metadata as **uninitialized**, thus generating **false positives**.

Before free: *free* itself will read chunk's bytes in order to perform some checks on metadata. So, if we reinitialize the shadow memory before it executes, *free* will see the memory location of the metadata as **uninitialized**, thus generating **false positives**.

By testing both the approaches with simple custom programs that just kept allocating

and deallocating chunks, neither of the methods appeared to be appreciably better than the other.

To solve this issue, we keep track of the write accesses performed during a call to *free* in a dedicated data structure. After *free* returns, we first reinitialize the shadow memory associated to the freed chunk and then we iterate over all the write accesses performed by *free* to set again the corresponding shadow memory as **initialized**. The number of write accesses performed by *free* inside the chunk is quite limited, so this operation does not require too much time, but it effectively reduces the number of reported **false positives**, as the subsequent calls to *malloc* will see the memory locations written by *free* as initialized.

Memory allocations preceding entry point

Sometimes, it is possible that calls to *malloc* happen before the entry point of the program is executed, usually due to some library initialization. But since we are interested in the overlaps generated during the execution of a program, *MemTrace* starts tracing memory accesses starting from the entry point. This causes mainly 2 problems: (1) if the program accesses some of the heap chunks allocated before the entry point execution, those accesses are always considered **uninitialized**, thus generating **false positives**; (2) if many chunks are allocated before the entry point execution (so that a whole memory page is completely allocated) and they are not traced by *MemTrace* and the program tries to access any of those chunks, a **segmentation fault** signal is thrown, because that memory page has no shadow memory associated.

As a solution, *MemTrace* **partially** keeps track of memory allocations executed before the entry point by setting as completely **initialized** any chunk that is allocated and resetting any chunk as **uninitialized** when it is freed. This tracing is *partial* because it does not really mirror the actual state of the memory, but it assumes any chunk as either initialized or uninitialized, regardless of the instructions actually accessing it. While this approach may ignore some uninitialized reads that may happen during libraries initialization, it's worth reiterate that we are interested in the overlaps generated by the program, and therefore the accesses executed before the program starts are out of the scope of *MemTrace*. So, this method allows us to deal with the mentioned problems, without affecting too much the overall execution time. More specifically, we completely solved the segmentation fault problem, because by keeping track of **all** the allocations, every allocated chunk will have a shadow memory associated; and we greatly mitigated the problem with the generated false positives, because all the chunks allocated during libraries initialization will be set

as **initialized** when the program accesses them.

Optimized strings functions

Modern processors often include vector extensions that allow it to execute **SIMD**[2] instructions. These instructions may be used to optimize portions of code that should repeatedly execute the same code on multiple data of the same type, and that are therefore usually implemented as a loop. By using vector instructions, indeed, it is possible to use a single instruction to simultaneously execute the same operation on a vector of similar data, thus reducing the number of iterations of the loop. In recent implementations of *glibc*[3] the same optimization technique has been used to optimize the functions related to string operations (e.g., *strlen, strcmp, strcat*, etc). Indeed, in the C language, strings are just an array of characters (i.e., *char*), and each *char* is represented by 1 byte. Instead of managing 1 byte at a time, the library leverages the SIMD instructions, if available, in order to manage a whole vector of bytes.

Consider, for instance, function *strlen*. This function simply scans the string from the address it accepts as a parameter looking for a string terminator (i.e., byte '\0') and returns the number of characters composing the string. The most straightforward way of implementing *strlen* consists in writing a *while* loop to scan each byte one by one and return when byte '\0' is found. The actual implementation does something similar, but, instead of considering one byte at a time, it considers a vector of 16 bytes at each iteration of the loop. SIMD instructions, indeed, allow to perform the following sequence of operations:

1. compare all the bytes of the vector to byte '\0' **simultaneously**.
2. extract a *bitmask* representing the result of the comparisons.
3. store the index of the first set bit in a register. This is the number of characters that compose the string.

With this sequence of operations, it is possible to find the string terminator, also reducing the number of iterations of the loop. Similarly, all the functions related to strings have been optimized. As a consequence, these functions may execute some **uninitialized reads** not really because they are caused by an error in the program, but simply because the applied optimizations make the used SIMD instructions read beyond the string terminator.

[2]Single Instruction Multiple Data
[3]*MemTrace* has been tested on Ubuntu with glibc-2.31 installed

Let's consider *strlen* again and assume its parameter is the string "*Lorem ipsum justo.*", which is 18 bytes long. The optimized version of *strlen* will load the first 16 bytes (bytes $0 \sim 15$) of the string (V1 in Figure 2.9) and compare them to byte '\0'. Since none of them is the terminator, the comparison will fail with all the bytes and therefore a new iteration begins. So *strlen* loads bytes $16 \sim 31$ (V2 in Figure 2.9) and compares them to byte '\0'. The string terminator is found at index 2 of this vector (starting from 0), and adding this index to the number of bytes already compared in previous iterations (16) we obtain the correct size of the string. Note, however, that the function had to load 32 bytes to compute the result, whereas the string was only 18 bytes long. This means that *strlen* loaded bytes beyond the end of the string, which are potentially **uninitialized**.

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V1	L	o	r	e	m		i	p	s	u	m		j	u	s	t

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
V2	o	.	\0	?	?	?	?	?	?	?	?	?	?	?	?	?

Figure 2.9: Loaded byte vectors from the example string. The reported offset is relative to the beginning of the string.

Every call to a string function can, therefore, generate **false positives** and, since these functions are widely used, the total amount of generated false positives may be very high. Since we are working with binaries, we don't have any information about the function being executed, so we cannot know when an uninitialized read is performed during the execution of a string function. In order to solve this issue, we implemented a quite simple but effective **heuristic**. When *MemTrace* detects an uninitialized read it assumes it is a false positive due to a string function if all the conditions listed in Table 2.2 hold simultaneously. Note that these conditions are evaluated on each loaded vector individually, and therefore conditions 2 and 3 may happen simultaneously on the same string. Let us better explain the meaning of conditions $1 \sim 3$.

Condition 1: the string is shorter than 16 bytes and its pointer does not satisfy the alignment required by the executed SIMD instruction (Figure 2.10a)

Condition 2: the string pointer does not satisfy the alignment required by the executed SIMD instruction, but it's long enough to fill the vector of bytes (Figure 2.10b)

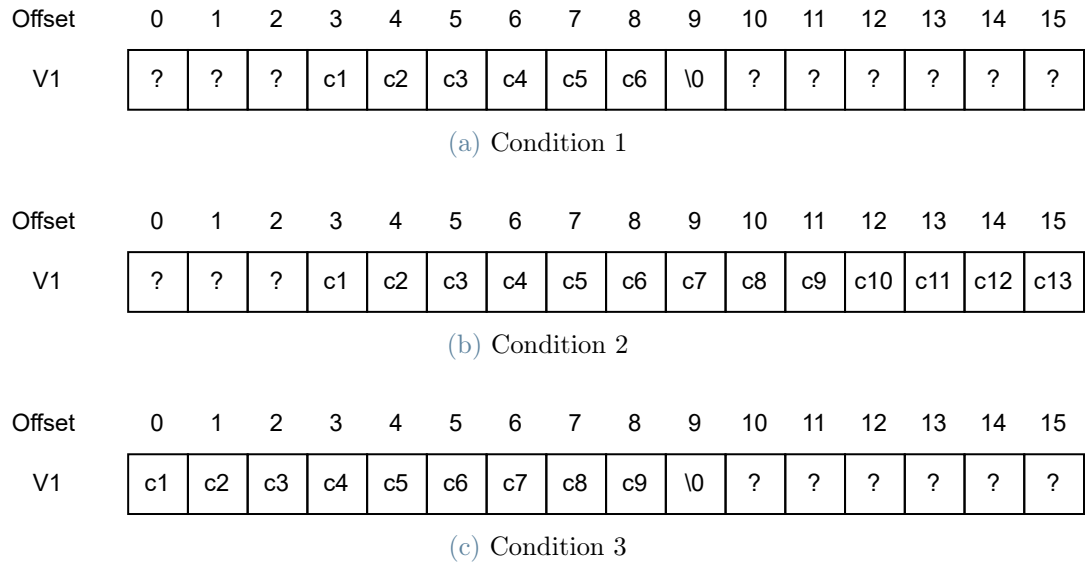
Condition 3: the string has a length which is not a multiple of the vector size, so when

the end of the string is loaded, the bytes beyond the string terminator are uninitialized (Figure 2.10c)

Condition	**Reason**
Access size is higher or equal to 16 bytes	Optimized string functions usually load vectors of bytes whose size is at least 16 bytes
The instruction is not a system call	Some system calls may access memory reading more than 16 bytes. For instance, a call to *write* may be incorrectly recognized as a false positive if we apply the heuristic also to system calls
The accessed memory location contains at least 1 **initialized** null byte (byte '\0', i.e., the string terminator)	Strings are usually terminated with a null byte written right after the last byte of the string. If there is no initialized null byte, it is not likely that the considered uninitialized read is due to string function optimizations
(1) There is more than 1 uninitialized interval **OR** (2) there is only 1 uninitialized interval, which begins at index 0 and ends **before** the first initialized null byte is found **OR** (3) there is only 1 uninitialized interval, which begins **after** the first initialized null byte is found	Conditions $1 \sim 3$ are used to recognize all the possible layouts for the bytes of a string

Table 2.2: Conditions for the string functions heuristic

With this heuristic, the previous example will not generate any false positive, because *MemTrace* will recognize the uninitialized read as part of an optimized string function (condition 3 is satisfied), thus ignoring it. Unfortunately, this is not enough to really mitigate the problem, because the actual optimizations applied to string functions sometimes also leverage **loop unrolling**. In some cases, indeed, the function will load more than 1



(a) Condition 1

(b) Condition 2

(c) Condition 3

Figure 2.10: Possible string layouts

vector at a time (usually 4), and then it will manage all of them one after another, before a new iteration of the loop begins, thus still possibly generating **false positives**. In order to clarify which is the actual problem, let us consider again the string "*Lorem ipsum justo.*", whose size is 18 bytes. If loop unrolling is not applied, as we said, no uninitialized reads are reported thanks to the heuristic. However, let's assume loop unrolling is applied and that 4 vectors are loaded (Figure 2.11). In this case, not only bytes $16 \sim 31$ from the string pointer may contain uninitialized bytes, but also the last 2 loaded vectors may be **completely uninitialized**.

Our heuristic requires to detect at least 1 initialized null byte to be triggered, so the uninitialized reads performed while loading the last 2 vectors of bytes are not recognized as part of a string function optimization, and therefore are not ignored. To fix this, we still leverage the heuristic. In fact, when it is triggered we assume that the program is probably executing an optimized string function. So, after the heuristic is applied once, we simply ignore **all** the **completely uninitialized** reads, as they are probably due to the loop unrolling optimization, until a *return* instruction is executed, meaning that the string function terminated. With this simple extension, our example does not generate any false positive even if loop unrolling is applied and, after some tests, the heuristic is actually able to ignore most of the uninitialized read performed due to the optimizations in string functions.

The issue, however, is not completely solved, as some cases may still generate **false positives** as well as **false negatives**. The first sources of false positives are the strings

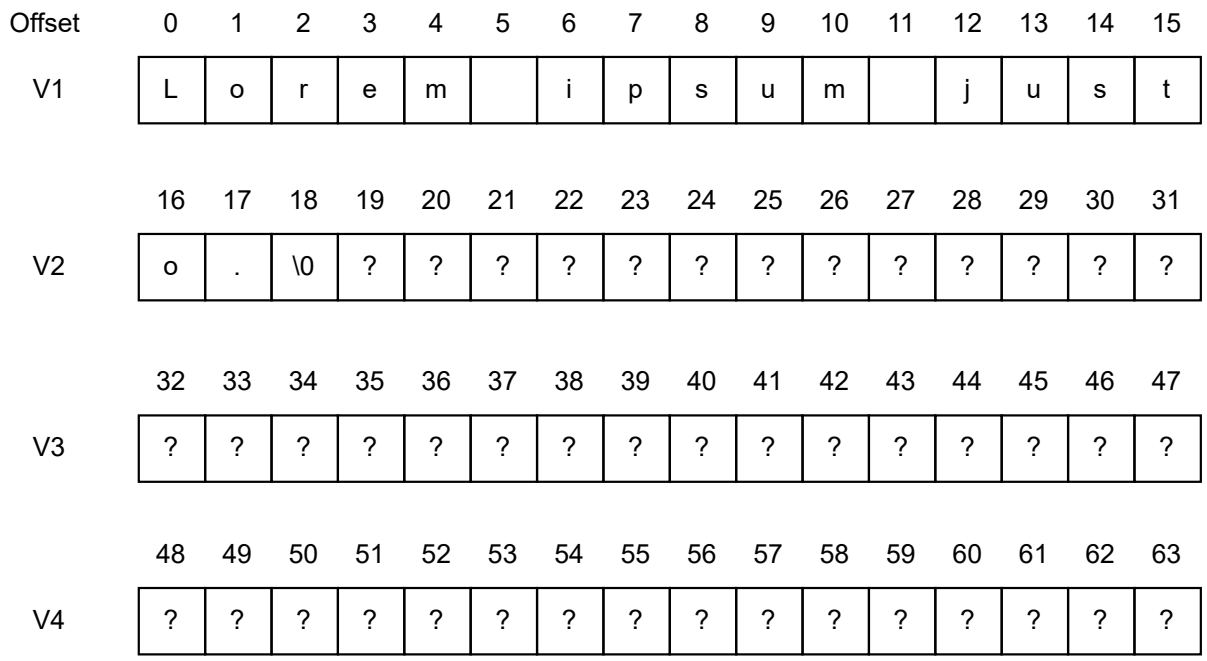


Figure 2.11: Loaded byte vectors from the example string with loop unrolling. The reported offset is relative to the beginning of the string.

whose size (terminator included) is equal to a multiple of the vector size. In these cases, indeed, the string terminator will be stored as the last byte of the vector. So, the whole vector will be **initialized**, and therefore will not trigger the heuristic. If loop unrolling is not applied, this should not be a problem, as the function should return as soon as the terminator is detected. But if it is applied, all the vectors loaded after the terminator may be completely uninitialized and, since the heuristic has not been triggered, they are not recognized as part of string function optimizations and therefore reported. This is, however, a quite rare situation, as it requires a strict condition to be verified. Moreover, our taint analysis helps by ignoring those vectors which are loaded into a register but never used by any instruction different from a *cmp* instruction or one of its variants, thus further reducing the number of reported false positives.

Another situation that may generate false positives is when the compiler performs **procedure inlining** of the string functions. This does not seem to happen very frequently, but when it happens, the memory accesses size is usually below 16 bytes, thus not satisfying the heuristic condition.

False negatives, instead, are mainly caused by optimizations performed by the compiler in other functions or portions of the program. Indeed, it is possible that the compiler uses the same set of SIMD instructions used to optimize string functions to optimize

also other loops dealing with arrays of data (e.g., an array of integer values). If any of those instructions satisfy the condition that triggers the string function heuristic, it will be ignored and may ignore all the subsequent loads of completely uninitialized vectors, if any. Unlike false positives, which are reported and can be manually verified through a debugger, we have no information available to try and verify false negatives. For this reason, it is not really possible to quantify or even qualitatively say how many of them are generated. To avoid possibly ignoring relevant uninitialized reads, however, we made the scope of applicability of the heuristic tunable. By default, it is applied only to instructions belonging to the libraries loaded by the executable, so that any optimization applied by the compiler to the code of the program will not be ignored. This, of course, does not solve the problem for other optimized functions implemented into libraries (e.g., *memcpy*), but, if needed, it is possible to completely disable the heuristic or even enable it also for binary's code.

Finally, we noticed that most of the uninitialized reads reported by instructions in optimized string functions were false positives. In an attempt to further reduce the reported false positives, we also implemented a post-process **filter** that makes use of *pyelftools* [17] to parse the debug information of the standard library (usually installed by default in operating systems) and filters out all the uninitialized reads performed by instructions belonging to string functions. Just like the string heuristic, also the filter can be disabled.

As we will better see in Chapter 4, the usage of the heuristic and the filter effectively reduce the number of reported false positives, still allowing to detect actual uninitialized reads throughout the executable.

Stack clash mitigation

Every process running on a computer divides its virtual address space into memory regions, or sections. One of these regions is used as the **stack**, which grows from higher addresses toward lower addresses. If the stack grows too much, it could collide (or clash) with other memory regions (e.g., the heap, the stack of other threads, etc..), thus possibly allowing an attacker to use stack variables to access other regions or, conversely, use objects stored in the colliding memory region to access the stack. As a mitigation technique, the operating system usually allocates a stack **guard page** right after the last page of the stack. This way, if a program tries to access any address in the guard page, an error is thrown, and execution is therefore interrupted. This is, however, not enough: if the program allows to choose somehow the growth of the stack (through a dynamic stack allocation) or any function's frame requires to allocate big portions of the stack, it

is possible to allocate more than 1 memory page of stack, thus possibly go beyond the guard page and therefore bypass the applied mitigation. For this reason, major compilers implemented a new mitigation technique, that, combined with the stack guard page, further reduces the possibility to exploit stack clash vulnerabilities.

The *stack clash mitigation* [4] implemented by compilers simply divides big stack allocations into a sequence of smaller allocations, so that it is never possible to bypass the guard page. More precisely, allocations that are bigger than a memory page are performed as explained by the pseudocode of Algorithm 2.1, whose schema is also depicted in Figure 2.12.

Algorithm 2.1 Stack clash mitigation

```
1: remaining_size = requested_allocation_size
2: while remaining_size >= PAGE_SIZE do
3:     SP = SP - PAGE_SIZE
4:     remaining_size = remaining_size - PAGE_SIZE
5:     probe_allocated_stack()
6: end while
7: SP = SP - remaining_size
```

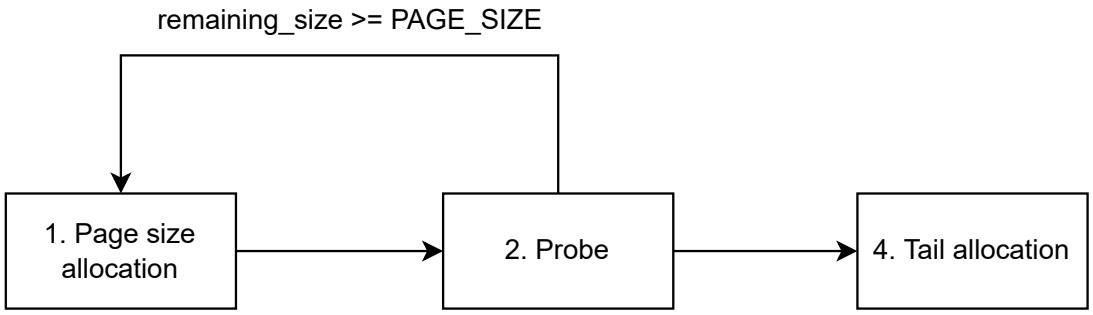


Figure 2.12: Stack clash mitigation scheme

The probe implementation is not standard, and therefore it can change according to the specific implementation of the compiler. In all the cases, however, it consists of a memory access to the newly allocated stack space. If the performed access is a read access, then it will be an **uninitialized read**, and it will be therefore reported by *MemTrace*. Since it is not so rare that a function requires a frame bigger than a memory page and functions are usually called more than once during program execution, this mitigation may generate many **false positives** if a read access is used as a probe (e.g., like in GCC).

In order to reduce the number of reported false positives due to stack clash mitigation, we implemented some analysis functions whose objective is simply to detect instructions

that perform stack allocations (which are simple *sub* instructions where the first operand is the *stack pointer*) and update an object representing the last executed stack allocation accordingly. If during the main analysis function we detect an uninitialized read that immediately follows a stack allocation whose size is equal to the page size and that is accessing any address of the just allocated stack space, then we assume it is due to the stack clash mitigation, and therefore ignore it. Notice that this does not completely solve the problem. Indeed, *stack clash mitigation* implementations may slightly differ from the scheme depicted in Figure 2.12. This usually can happen when dynamic stack allocation is used. In these cases, the compiler may add a probe *before* a stack allocation or *after* a tail allocation. Both these kinds of probes are not distinguishable from actual and possibly relevant **uninitialized reads**, and therefore cannot be ignored, thus generating a false positive. Notice, however, that this situation is quite rare, and therefore the number of reported false positives is very limited.

Finally, if the program allocates **exactly** a page size of stack, and it is compiled without the *stack clash mitigation*, it is possible that *MemTrace* will ignore the uninitialized read executed immediately after the stack allocation, if any, thus generating a **false negative**. However, this situation is extremely rare, as it requires a very strict condition to be verified, thus making the number of possibly generated false negatives very low.

Propagation of bytes state

As explained in Section 2.2, we implemented a *taint analysis* in order to keep track of uninitialized bytes and their *origin*. To do that, *MemTrace* must be able to propagate the state of bytes from source to destination whenever a data transfer instruction is executed. The difficult thing about this task is that every instruction has its own semantic and many of them may use registers of different types or sizes as source and destination or may access a memory location whose size is different from the size of the registers used as operands. Therefore, it is required to know how a specific instruction transfers bytes. Of course, given the huge amount of instructions and the complexity of their semantics, it is not feasible to use a single huge *if* or *switch* statement to manage every single instruction the program may use. So, we implemented an **instruction manager**, which is responsible for checking the instruction the program is going to execute and call the correct **instruction handler**.

This component is quite similar to the *system call manager*: the *instruction manager* works as an intermediary, exposing a simple and common interface to the other compo-

nents (Figure 2.13) and, according to the *opcode*[4] it receives as a parameter, it will relay the request to a specific *instruction handler*, which models the data transferring behavior of the actual instruction. In order to make *MemTrace* easily extensible, we divided the specific handlers in 2 major classes, as shown by Figure 2.13:

Mem instructions: instructions reading/writing bytes from/to memory (i.e., *loads* and *stores*

Reg instructions: instructions simply copying bytes from a set of registers to another one

To reduce the number of required specific handlers, we also implemented a *default* handler for each class of data transfer (i.e., load, store, propagate) which allow to correctly handle most of the instructions.

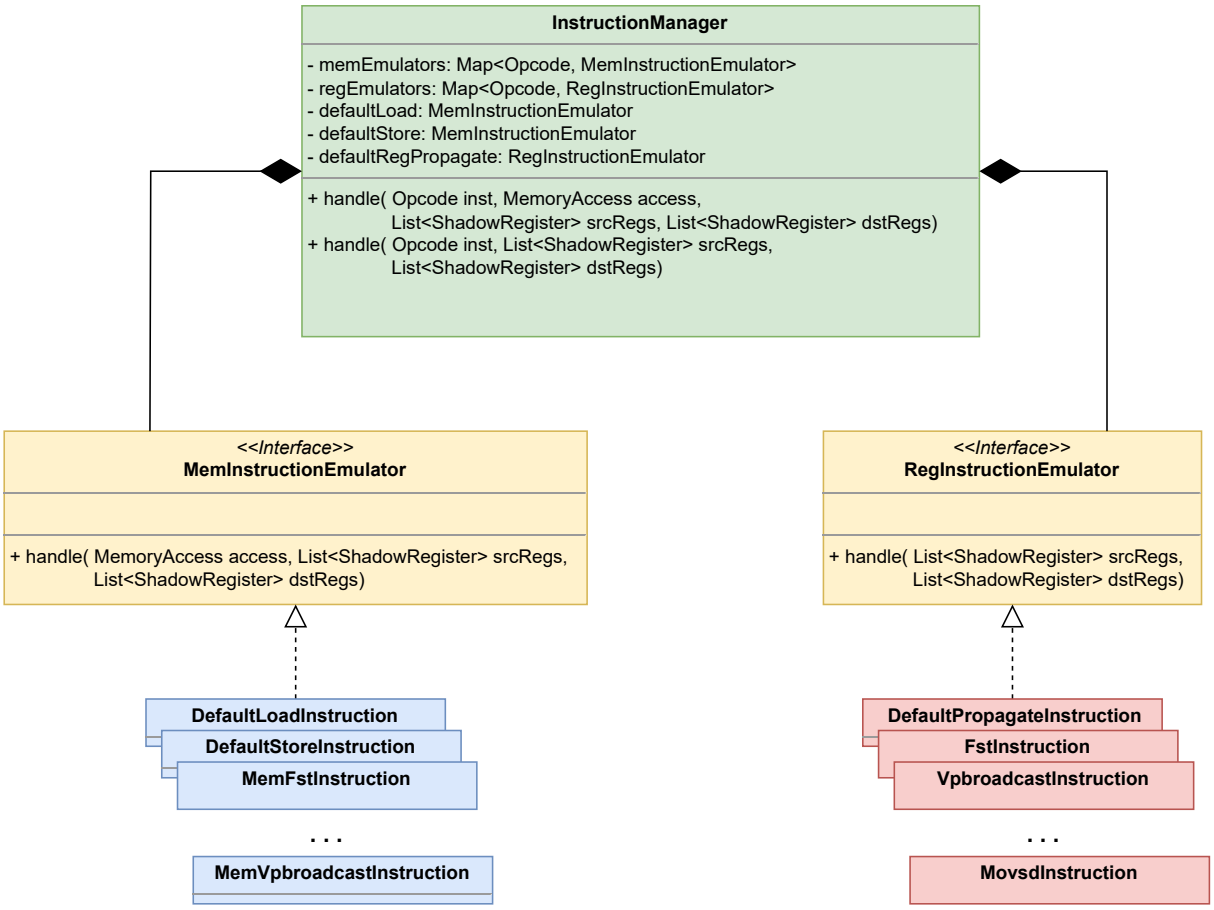


Figure 2.13: Conceptual class diagram of the Instruction Manager

To try and understand which instructions required a specific handler, we made the *instruc-*

[4]Portion of an instruction uniquely associated to the instruction itself and therefore allowing to recognize it.

tion manager create a secondary report which simply contains a list of all the instructions it receives where source and destination registers have a different size or the accessed memory location has a size different from the size of the registers used as operands. Then, we launched *MemTrace* with the whole suite of utilities from *Coreutils* and other programs used for tests and validation, and we **manually** scanned the reports and verified whether each reported instruction was correctly handled by the *default* handlers or not. If it was not, the instruction required a specific handler. Using this approach, we implemented a total of 12 specific instruction handlers. Of course, it is possible we missed some instructions requiring a specific handler, and therefore there might be some incorrectly handled instructions. However, we will see in Chapter 4 that the implemented handlers seem to be enough to correctly detect *uninitialized reads*.

Finally, notice that if any instruction has no specific handler and has not been verified as correctly handled by the default handlers, it is reported in the secondary report created by the *instruction manager*, thus allowing the user to be aware of the possibly missing handler, which can be easily implemented and added to extend the tool with a minimum amount of changes to the existing source code.

Zeroing XOR

Table 2.3 reports the truth table of the *XOR* operator. As it depicts, if the *XOR* is applied to 2 bits with the same value, the result is always 0. This is true also if we extend the *XOR* operator to be a bitwise register operator. Indeed, the bitwise *xor* instruction executes a *xor* of the bits from the source registers that occupy the same position i, and stores the result at position i of the destination register (that coincides with the first source register), as shown by Figure 2.14. So, if we execute a *xor* instruction using the same register twice as operands, we are essentially setting that register to 0. Moreover, in x86 architectures, a *xor* instruction is smaller than a *mov* instruction used to load an immediate into a register with size bigger than 1 byte. For this reason, often compilers leverage the *xor* instruction as an optimization to set a register to 0.

$\oplus$	**0**	**1**
0	0	1
1	1	0

Table 2.3: Truth table of *XOR* operator

When this happens, if the register that is being zeroed contains an uninitialized byte, the

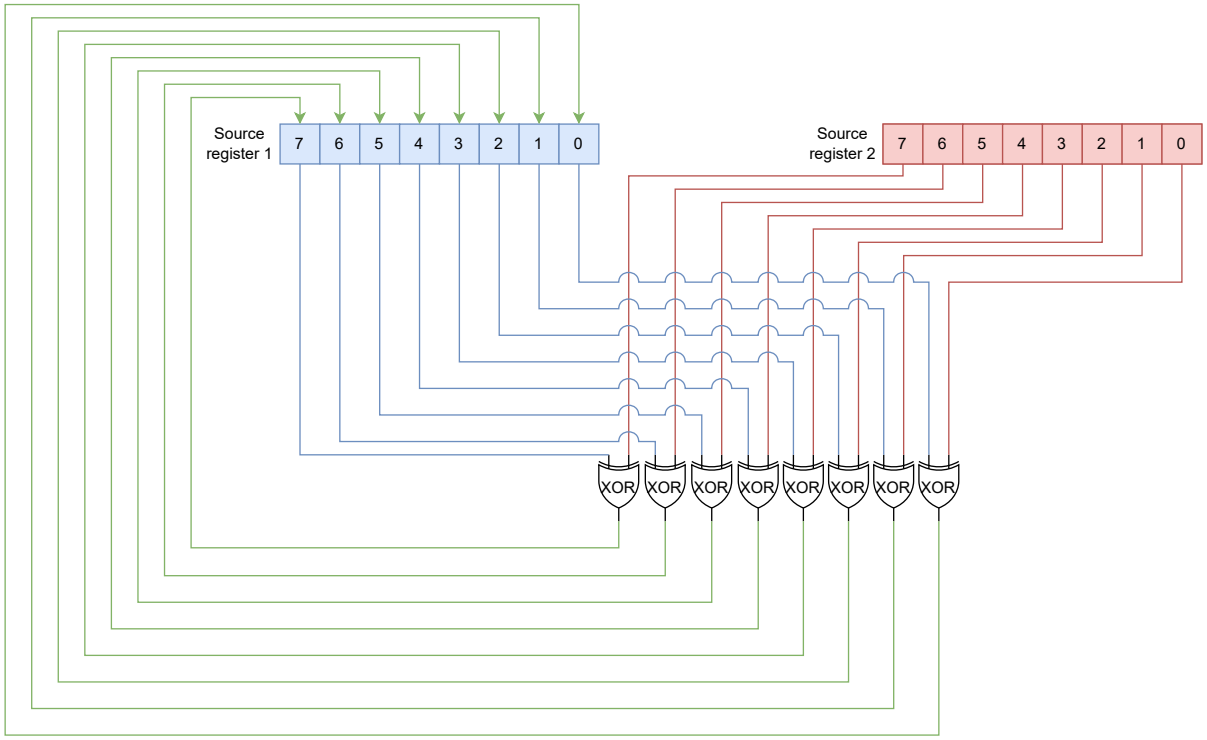


Figure 2.14: Bitwise *xor* instruction schema for 8 bit registers

taint analysis will detect the uninitialized byte usage, and therefore will report its origin. While it is true that the uninitialized byte is being read and used by an instruction, in this case the instruction is simply used to set to 0 that register, thus overwriting the register's content. Therefore, in these cases, we don't want *MemTrace* to consider the instruction as a usage of uninitialized bytes. The solution to this problem is quite simple. Indeed, it is sufficient to perform a check on the instruction before calling the taint analysis functions. If it is a **zeroing *xor***, function *checkSourceRegisters*, which is the one that reports the corresponding uninitialized reads if uninitialized bytes are detected inside any source register, is simply skipped. Note that function *checkDestRegisters* is still executed, so that the destination shadow register is updated coherently, thus setting all of its bits as initialized, because the zeroing xor is setting the destination register to 0.

In order to detect zeroing xors, *MemTrace* performs the checks reported in Algorithm 2.2: first, it verifies that the opcode of the instruction being executed is the opcode of a XOR instruction; then, it checks the number of source and destination registers and, finally, it checks whether the source registers are the same register. This way, all the uninitialized bytes usages due to zeroing xors are ignored by *MemTrace*, thus reducing the number of false positives.

Finally, note that while the xor instruction can use memory operands as source or des-

tination, it is not possible to have a zeroing xor acting directly in memory, because x86 ISA does not support xor instructions using 2 memory operands as sources.

Algorithm 2.2 Zeroing xor recognition

```
1: procedure isZeroingXor(inst):
2:     if inst.opcode == XOR_OPCODE and inst.srcRegs.size == 2 and
       inst.dstRegs.size == 1 and inst.srcRegs[0] == inst.srcRegs[1] then
3:         return true
4:     end if
5:
6:     return false
7: end procedure
```

XSAVE and XRSTOR

According to [27], x86 architectures use *xsave* and *xrstor* instructions in order to store and restore, respectively, a full or partial copy of the processor state components at the memory location specified by the address used as an operand. In practice, they are used to store and restore a copy of the processor's registers in memory. If *xsave* stores a register containing some uninitialized bytes, and then the program overwrites that register, we would like that *MemTrace* was able to restore the uninitialized state of the register when *xrstor* is called, otherwise the state of the shadow register would not reflect the state of the corresponding architectural register.

Unfortunately, while *Intel PIN* is able to detect when these instructions are used, it does not keep track of which registers are copied/restored. Indeed, the instructions do not accept an explicit set of register operands, but they use some control registers in order to select them. This means that *MemTrace* cannot rely completely on the information the underlying DBI framework makes available about *xsave* and *xrstor* instructions. For this reason, we implemented a specific analysis function which is invoked by *MemTrace* when either *xsave* or *xrstor* is executed. Based on the description from [27], we implemented an analysis function that reads the control registers to verify which registers are involved and mimics the behavior of *xsave* and *xrstor*. Notice that the actual implementation of these instructions also copies and restores the values stored in the control or status registers of the processor. Those registers hold values used by the processor to change or control its internal behavior or to evaluate conditions for branches. So, they are not directly accessible by the program and therefore they cannot contain bytes loaded by an uninitialized read memory access. For this reason, *MemTrace* does not trace them, and therefore the xsave/xrstor handler does not take them into consideration. This means

that the analysis function only partially models the actual behavior of the instructions, thus storing and restoring the state only for the registers which are used by the program to hold temporary values (e.g., integer, floating point or vector registers). By using this additional analysis function, *MemTrace* is able to correctly handle *xsave* and *xrstor* in case any register was uninitialized.

3 | Fuzzing

MemTrace is a dynamic analysis tool and, as such, has an intrinsic limitation: it can only report overlaps detected in the execution path the program traverses. In order to partially deal with this limitation, we combined *MemTrace* with *AFL++* [20], which is one of the most effective and widely used fuzzers. By fuzzing the binary, AFL++ will generate a lot of inputs, that, as shown by Figure 3.1, can be subsequently used as an input for the program executing it through *MemTrace*.

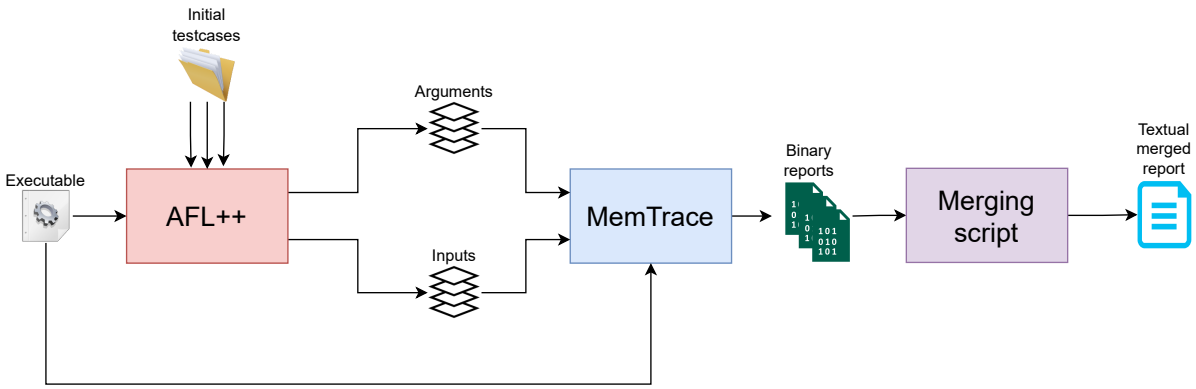


Figure 3.1: Block diagram of *MemTrace* combined with *AFL++*

AFL++ is designed to try and explore as many paths as possible and it will store only those inputs that it considers *interesting*, which are, mainly, the ones crashing the program and the ones which explore new paths w.r.t. previous executions. So, by using the inputs generated by the fuzzer to execute the program with *MemTrace*, we are able to explore more execution paths. However, a single execution of a program with *MemTrace* will generate a single binary report file, containing information about the overlaps detected in that specific path. By executing *MemTrace* once for each input generated by the fuzzer, it will generate as many binary reports, and even if we passed them all through the parser to generate the textual report, they would be too many to be manually checked and extract some useful results. So, we implemented a merging script that uses the binary report *parser* to extract information about overlaps from every binary report generated during the combined execution of AFL++ and *MemTrace* and then merges them all in a **merged**

report which collects, in a single textual file, the summary of the results obtained in all the executions.

The *merged report* is composed by a sequence of *tables*, one for each instruction performing an uninitialized memory read. This means that if 2 different executions of the program executed the same instruction performing an uninitialized read, those *overlaps* will be merged and reported in the same table. Notice, however, that if ASLR is enabled in the system, different executions of the program will load the same instruction at a different absolute address. So, in order to recognize the table a certain instruction should be reported in, we need to consider the relative offset of the instruction from the base of its memory region. This way **all** the instructions with the same offset from a specific memory region are reported in the same table.

The objective of *MemTrace* is to report *overlaps*, that we defined as a set composed by an *uninitialized read* and an *access set*, i.e., the set of instructions that wrote the bytes it read. So, each *table* is composed by more *overlaps*. In order to avoid confusion and redundancy, identical *overlaps* are not reported twice, but each of them also contains a counter reporting the number of inputs that were able to generate the considered overlap, giving the user some insight about how difficult it is to generate such an overlap. Given 2 *overlaps*, they are considered **different** if (a) they have different *access sets*, that is, the instructions that wrote the bytes read by the uninitialized read are different; or (b) their uninitialized reads have different *uninitialized intervals*, i.e., the interval of indices containing uninitialized bytes; or (c) the uninitialized reads access different *memory locations*. Again, in order to deal with ASLR, *memory locations* are reported as couples composed by an offset from the beginning of the memory region they belong to (e.g., the heap) and a size. Figure 3.2 shows an example of tables reported in a merged report, highlighting the type of information it contains and where it can be found.

Besides helping explore more execution paths, the fuzzer is helpful also because it allows detecting those uninitialized reads that can be somehow controlled. Indeed, if we want to exploit an uninitialized read to obtain a leak, we would need to control either the content of the memory location it reads from or the memory location itself. So, it would be useless to report uninitialized reads that always read from the same memory location and always read the same bytes. *MemTrace* does not actually consider the content of memory locations (i.e., the actual bytes read), but if a certain table in the merged report only contains a single overlap, it is likely not to be useful to obtain a leak, and therefore, we can avoid reporting it, thus making the merged report much clearer. Moreover, all the overlaps with an empty access set, which are the ones composed of an uninitialized read reading a memory location that has **never** been written by the program since its beginning, are

not reported at all. This is done because those uninitialized reads will read unpredictable bytes from memory, and *MemTrace* cannot, therefore, give any information about the origin of the bytes read. However, if needed, *MemTrace* can be easily configured to report both single overlap tables and empty access set overlaps by using specific command-line options.

Since *MemTrace* analyzes executable binaries, it cannot really distinguish which uninitialized reads actually lead to information disclosure. However, by inspecting the merged report and analyzing its overlaps (e.g., through a debugger) the user can see which are the uninitialized reads in the program that can lead to a leak and the reported overlap sets are helpful to understand **if** and **how** it is possible to control them to get **arbitrary** information.

Finally, notice that, as a side effect, *MemTrace* also helps detect other types of vulnerabilities involving an uninitialized read. For instance, if a program allows a user to write from *stdin* in a certain buffer and during execution it happens that the buffer overlaps an uninitialized function pointer, *MemTrace* will report the uninitialized read caused by the call to the function pointer and, analyzing the report, it is easy to see that it's possible to **directly** control the target of the *call* instruction from *stdin*, thus allowing to execute arbitrary code. We'll describe more in depth a case of such an example in Chapter 4.

Command-line arguments fuzzing

Sometimes the command-line arguments passed to a program may change program's behavior, thus making the execution traverse paths that would not be taken otherwise. In order to deal with this, we leveraged the fuzzer once again. Extending an example provided within AFL++'s repository [1], we implemented a library for AFL++ that allows to fuzz also the arguments passed to the fuzzed program. The library simply implements a **hook** for function $\_\_libc\_start\_main$, which is responsible to prepare and call function *main* during a program's startup. So, the call to $\_\_libc\_start\_main$ is intercepted and, before calling the original version of the function, some of the bytes generated by the fuzzer are used as a command-line argument, updating *argc* and *argv* accordingly. Finally, the original $\_\_libc\_start\_main$ is called, passing it the new values of *argc* and *argv*. The generated arguments are stored and, as depicted in Figure 3.1, they are subsequently used to execute again the same binary within *MemTrace*. This way, the fuzzer may help traverse new paths in the executable, thus increasing the coverage. Of course, fuzzers are not really designed to fuzz command-line arguments, so it is possible that many inputs generated with argv fuzzing enabled pass to the program invalid arguments, which

may cause the program to exit early. However, as we will better see in Chapter 4, this approach is good enough to find some *vulnerabilities* which are triggered only if some specific command-line arguments are used. Obviously, it comes with some **drawbacks**, which will be explained in Chapter 6.

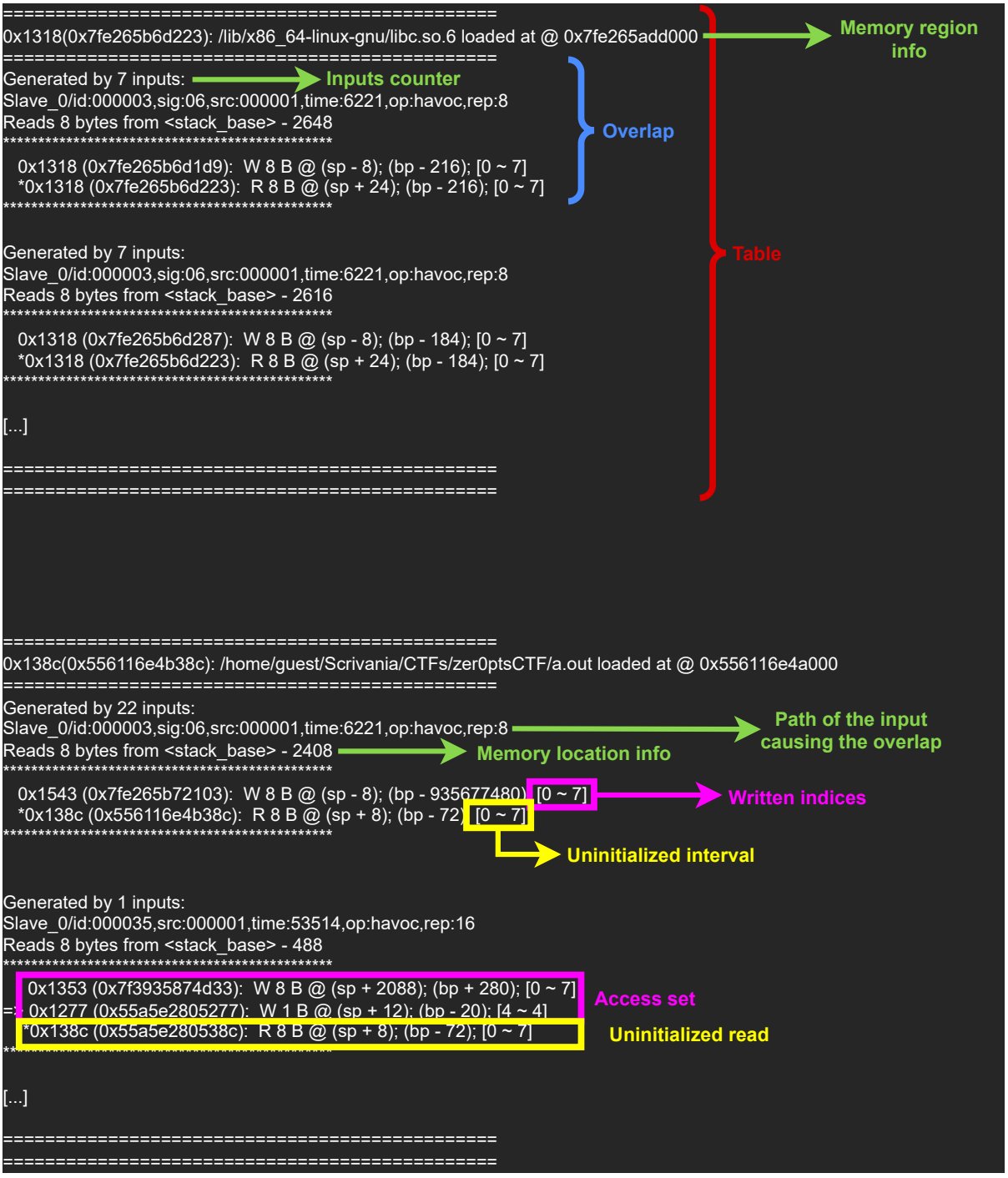


Figure 3.2: Portion of a *merged report*

4 | Validation

We performed 2 types of tests:

Timing: since we want to use *MemTrace* in combination with a fuzzer, execution time is a major concern, because if the overhead introduced by the analysis to the execution of a program was too heavy, it would be infeasible to execute it with every generated input.

Functional: of course we want to verify *MemTrace* works well and is actually able to reach the objectives it is designed for.

4.1. Timing tests

As a first obvious attempted method to compute the overhead of *MemTrace*, we collected the time required to execute a program and the time required to execute the same program (with the same parameters and inputs) within *MemTrace*. However, we noticed that most of the time consumed by *MemTrace* was actually spent by the underlying DBI framework to load the executable and all the libraries it requires, so we decided to try and isolate the overhead introduced by the actual analysis by the one introduced by *Intel PIN*. In order to compute the analysis overhead introduced by *MemTrace* we required to collect the following information for a certain program P:

$\langle NullTime \rangle$**:** time required to execute program P within *Intel PIN*, but with **no analysis functions** registered.

$\langle InstrumentedTime \rangle$**:** time required to execute program P within *MemTrace*

We collected the required data for the utilities from *Coreutils* and computed the overhead of the analysis as a **multiplication factor** as

$$\langle Overhead \rangle = \frac{\langle InstrumentedTime \rangle}{\langle NullTime \rangle} \tag{4.1}$$

As shown by Table 4.1, *MemTrace* analysis is, on average, about 8 times slower than the

execution with no analysis. In order to enable a comparison, since it is also implemented as a dynamic binary instrumentation tool, we did the same thing by collecting the execution times with and without registered analysis using *Memcheck* as an analysis tool. Computing the analysis overhead for *Memcheck* using Equation (4.1) again, we found out that *Memcheck* analysis is, on average, about 2 times slower than execution with no analysis (Table 4.2) and, therefore, that *MemTrace* is about only 4 times slower than *Memcheck*. Considering the additional amount of analysis performed by *MemTrace* on **every** memory access, this is quite a reasonable and satisfying result.

Note, however, that the multiplication factors we computed for both *MemTrace* and *Memcheck* are not constant. Indeed, being both of them dynamic binary instrumentation tools, the actual execution time highly depends on several variables, such as the length of the input, the number of executed instructions, the number of memory accesses, the number of executed system calls, etc... In any case, the comparison can still be considered valid, as it was performed by comparing the same programs executed using the same inputs, thus fixing most of the variables mentioned above.

Begin of Table 4.1

Utility	**NullTime(ms)**	**InstrumentedTime(ms)**	**Analysis overhead**
dd	753	6265	8.32
b2sum	553	4393	7.94
base32	542	3939	7.26
base64	534	4004	7.49
basename	514	3675	7.14
cat	457	3574	7.82
chgrp	675	4815	7.13
chmod	511	3753	7.34
chown	709	4896	6.90
cksum	534	3949	7.39
sort	637	5173	8.12

Continuation of Table 4.1

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
comm	535	4179	7.81
cp	589	4684	7.95
csplit	597	4500	7.53
cut	550	4023	7.31
date	602	4524	7.51
df	865	7342	8.48
dir	694	5338	7.69
dircolors	525	3880	7.39
dirname	510	3631	7.11
du	1089	12326	11.31
echo	498	3669	7.36
env	531	3890	7.32
expand	534	3894	7.29
expr	569	4043	7.10
factor	514	3763	7.32
fmt	699	5231	7.48
fold	671	5204	7.75
getlimits	75	71	.94
groups	708	4802	6.78
head	668	5205	7.79
hostid	770	5385	6.99
id	926	7772	8.39

Continuation of Table 4.1

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
join	666	5425	8.14
link	443	3380	7.62
ln	447	3426	7.66
ls	802	6804	8.48
md5sum	504	4119	8.17
mkdir	552	4200	7.60
mkfifo	532	4307	8.09
mknod	911	5070	5.56
mktemp	478	4090	8.55
mv	561	4426	7.88
nice	408	3148	7.71
nl	551	4223	7.66
nproc	513	3802	7.41
numfmt	581	4463	7.68
od	643	5219	8.11
paste	502	3892	7.75
pinky	663	5529	8.33
printenv	533	3871	7.26
printf	563	4103	7.28
ptx	803	6501	8.09
pwd	529	3767	7.12
readlink	511	3666	7.17

Continuation of Table 4.1

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
realpath	523	3823	7.30
rm	71	71	1.00
rmdir	473	3478	7.35
seq	8867	31139	3.51
sha1sum	543	4461	8.21
sha224sum	557	4651	8.35
sha256sum	547	4831	8.83
sha384sum	567	4830	8.51
sha512sum	562	4763	8.47
shred	531	3978	7.49
shuf	578	4426	7.65
sleep	1445	4783	3.31
split	638	5319	8.33
stat	1032	8595	8.32
stty	675	5636	8.34
sum	495	3929	7.93
tac	471	3742	7.94
tail	670	5356	7.99
touch	480	3546	7.38
truncate	631	4924	7.80
tty	506	3697	7.30
uname	492	3693	7.50

Continuation of Table 4.1

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
unexpand	510	3946	7.73
uniq	554	4087	7.37
unlink	463	3472	7.49
uptime	799	6492	8.12
users	490	3399	6.93
vdir	1231	10103	8.20
wc	589	4362	7.40
who	491	3669	7.47
whoami	690	4762	6.90
Average			7.45

End of Table 4.1

Table 4.1: *MemTrace* timing test

Begin of Table 4.2

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
dd	479	935	1.95
b2sum	443	841	1.89
base32	429	806	1.87
base64	433	816	1.88
basename	430	785	1.82
cat	441	799	1.81
chgrp	480	857	1.78
chmod	423	800	1.89
chown	483	849	1.75
cksum	434	793	1.82
sort	454	889	1.95
comm	447	826	1.84
cp	471	892	1.89
csplit	433	837	1.93
cut	436	847	1.94
date	449	845	1.88
df	519	954	1.83
dir	474	916	1.93
dircolors	428	816	1.90
dirname	436	780	1.78
du	738	1441	1.95
echo	450	779	1.73

Continuation of Table 4.2

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
env	450	833	1.85
expand	470	819	1.74
expr	445	846	1.90
factor	439	835	1.90
fmt	470	847	1.80
fold	472	864	1.83
getlimits	6	13	2.16
groups	450	869	1.93
head	462	873	1.88
hostid	474	932	1.96
id	523	994	1.90
join	423	792	1.87
link	425	776	1.82
ln	413	774	1.87
ls	550	1004	1.82
md5sum	438	817	1.86
mkdir	471	866	1.83
mkfifo	458	859	1.87
mknod	462	851	1.84
mktemp	424	798	1.88
mv	455	860	1.89
nice	402	740	1.84

Continuation of Table 4.2

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
nl	470	828	1.76
nproc	454	791	1.74
numfmt	435	829	1.90
od	444	866	1.95
paste	444	806	1.81
pinky	464	875	1.88
printenv	449	790	1.75
printf	421	803	1.90
ptx	480	938	1.95
pwd	390	755	1.93
readlink	388	737	1.89
realpath	391	734	1.87
rm	5	6	1.20
rmdir	420	731	1.74
seq	6838	7370	1.07
sha1sum	440	1179	2.67
sha224sum	493	915	1.85
sha256sum	564	913	1.61
sha384sum	416	826	1.98
sha512sum	420	827	1.96
shred	396	760	1.91
shuf	430	824	1.91

Continuation of Table 4.2

Utility	NullTime(ms)	InstrumentedTime(ms)	Analysis overhead
sleep	1414	1797	1.27
split	451	864	1.91
stat	577	1107	1.91
stty	466	903	1.93
sum	455	838	1.84
tac	453	836	1.84
tail	480	855	1.78
touch	392	815	2.07
truncate	442	866	1.95
tty	426	893	2.09
uname	402	769	1.91
unexpand	428	794	1.85
uniq	488	784	1.60
unlink	536	951	1.77
uptime	511	1047	2.04
users	404	789	1.95
vdir	582	1157	1.98
wc	433	845	1.95
who	407	796	1.95
whoami	464	850	1.83
Average			1.86

Continuation of Table 4.2			
Utility	**NullTime(ms)**	**InstrumentedTime(ms)**	**Analysis overhead**
End of Table 4.2			

Table 4.2: Memcheck timing test

4.2. Functional tests

4.2.1. Goals

MemTrace can be used either as a standalone tool or in combination with a fuzzer to try and explore more execution paths of the binary under analysis. For this reason, we set up 2 different types of functional tests, each with its own goal.

Uninitialized reads detection

In order to be able to report memory overlaps, *MemTrace* must obviously be able to detect **uninitialized read** accesses. To test its capability to detect and report uninitialized reads, we used some *real-world* binaries and binaries from some past CTF competitions with **well-known** vulnerabilities, so that we could easily verify the obtained results, and we executed them through *MemTrace* using the command-line arguments and/or the inputs that triggered the known vulnerabilities.

Fuzzing efficacy

Once we verified the capability of *MemTrace* to report uninitialized reads, we were interested in figuring out whether its combined execution with *AFL++* was effective. To do this, we leveraged the very same binaries used for the validation of *uninitialized reads detection* capability. This time, however, we wanted to know if the fuzzer was able to generate the inputs and command-line arguments required to trigger the known vulnerabilities in the binaries, so we used, as initial testcases for the fuzzer, some random or simple inputs that did not trigger the vulnerability, and we used the generated inputs and arguments to analyze the binary through *MemTrace*.

4.2.2. Dataset

We tested *MemTrace* with a set of binaries having known vulnerabilities. More specifically we tested the tool with the binaries listed in Table 4.3. Note that we compiled CGC binaries from `https://github.com/GrammaTech/cgc-cbs`, which is a porting of the CGC binaries to Linux.

4.2.3. Experimental setup

Uninitialized reads detection

This experiment has been performed on a local machine running Ubuntu, with *glibc-2.31* installed and very limited resources, namely 4 GB of memory and 2 CPUs. For each binary listed in Section 4.2.2, we crafted an input triggering the known *uninitialized read* vulnerability, and we simply launched *MemTrace* executing the binary with the crafted input. Then, we generated the textual report through the binary report parser and we **manually verified** all the reported overlaps, i.e., we verified that the reported *read* accesses were actually uninitialized and verified that the bytes they read had actually been written by the *write* accesses reported in their access set. Each execution required just a few seconds to terminate, including the time consumed by the underlying DBI framework to load the executable and start execution.

Fuzzing efficacy

Since fuzzing is a quite heavy task and may work better when more instances run in parallel, we performed this experiment in a docker container on a remote machine with many more resources available, namely 40 CPUs and about 370GB of memory. The container was run with an Ubuntu image, with *glibc-2.31* installed. As a dataset, we used again the same binaries listed in Section 4.2.2. This time, however, we used *MemTrace* combined with *AFL++* to verify whether the combined execution is actually able to *automatically* identify vulnerabilities in a program without requiring the user to manually craft a triggering input. For each binary listed in Section 4.2.2, we ran 4 parallel instances of the fuzzer for 8 hours using, as initial testcases, only simple or random inputs. When possible, we even enabled arguments fuzzing. In the end, a merged report was generated for each tested binary by using the merging script and we manually verified them similarly as we have done for the previous tests.

4 *real-world* binaries	**cbor2json** from OOCBORRT `https://github.com/objsys/oocborrt` Commit de254ab4193f6b9d865213a35482b65b7be01dee CVE-2020-24753
	md2html from MD4C `https://github.com/mity/md4c` Commit aa654230915db7439eb22ae8b0d6c58f4409e17d CVE-2021-30027
	cp from Coreutils-6.9.90 Vulnerability description: `https://github.com/coreutils/coreutils/blob/master/NEWS`, line 3294 Fix commit: a54e8bb8a547c2ee9147865e2eb42eece69e8072
	tail from Coreutils-7.6 Vulnerability description: `https://github.com/coreutils/coreutils/blob/master/NEWS`, line 2551 Fix commit: fc4d3f63b0c64992014b035e8b780eb230e0c855
3 binaries from *Capture The Flag* (CTF) competitions	**Contacts** from *picoCTF* 2018
	Full Protection from *ASIS CTF* 2020
	Stopwatch from *zer0ptsCTF* 2021
5 binaries from *Cyber Grand Challenge* (CGC)	**Accel** - KPRCA_00013
	TextSearch - KPRCA_00036
	HackMan - KPRCA_00017
	TFTTP - NRFIN_00012
	SSO - NRFIN_00033

Table 4.3: Validation dataset

4.3. Results

4.3.1. Uninitialized reads detection

For almost all the binaries in the dataset, *MemTrace* successfully identified the known vulnerability, thus reporting the instruction performing the uninitialized read and all the write accesses, if any, whose bytes have been read by the read access. The only binary in the dataset for which *MemTrace* was not able to report the vulnerability has been *md2html*. Listing 5 shows the portion of source code of *md2html* that contains the vulnerability.

```
/*
** From src/md4c.c.
** The last part of the if condition (namely "CH(off) == _T('#')") is
    comparing 2 characters and is responsible for the uninitialized read.
     Indeed, under certain conditions, off can be increased higher or
    equal to the size of the document content.
*/

[...]

/* Check for ATX header.*/
    if(line->indent < ctx->code_indent_offset  &&  CH(off) == _T('#')) {
        unsigned level;
        if(md_is_atxheader_line(ctx, off, &line->beg, &off, &level)) {
            line->type = MD_LINE_ATXHEADER;
            line->data = level;
            break;
        }
    }

[...]


```

Listing 5: *md2html* source code (*CVE-2021-30027*)

By observing the source code and referring to *CVE-2021-30027*, the vulnerability lies in the comparison of 2 characters. In the executable, this comparison is performed by using a *cmp* instruction. As explained in Chapter 2, *MemTrace* does not report the uninitialized read if the read bytes are used only by a *cmp* instruction, because it surely cannot lead to information disclosure. For this reason, *MemTrace* did not report the known vulnerability in *md2html* and, since it is due to an aware design decision of the tool, it is not really to be considered as a false negative.

For most of the tests, as shown by Table 4.4, **no false positives** were generated and,

when there were some, they were very limited in number and very easily recognized by using a simple debugger.

Finally, notice that some vulnerabilities may cause the program to perform more than a single uninitialized read. This happens, for instance, in *cp* from *Coreutils-6.9.90*. In this binary, the vulnerability consists in the usage of an uninitialized struct object. These structures are usually not accessed as a whole, but the program accesses their fields when it is required. So, every time the program accesses a different field of the uninitialized structure, it is performing an uninitialized read, which is detected and reported by *MemTrace*.

Binary	Uninitialized Reads	FP
cbor2json	1	0
md2html	0	0
cp	8	1
tail	2	1
contacts	1	0
full_protection	1	0
stopwatch	2	0
accel	3	0
textsearch	1	0
hackman	2	0
tfttp	1	0
sso	1	0

Table 4.4: Uninitialized reads detection tests results

We tried to compare the results obtained by *MemTrace* with the similar existing tools we described in Section 1.1. Unfortunately, *Sleak* is not publicly available, and the same is true for most of the binaries listed in its validation dataset, which makes the comparison impossible. So, we could only compare *MemTrace* with *Memcheck*, as we did with the timing tests. To enable the comparison, we performed the same set of tests (i.e., same binaries and same inputs) using *Memcheck* as an analysis tool. Table 4.5 shows the number of reported items for both *MemTrace* and *Memcheck*.

In most of the cases, *MemTrace* reports a lower number of items than *Memcheck*. This is because *MemTrace* only focuses on usages of uninitialized bytes, while *Memcheck* reports

binary	MemTrace	Memcheck
cbor2json	1	3
md2html	0	4
cp	8	4
tail	2	3
contacts	1	4
full_protection	1	5
stopwatch	2	137
accel	3	12
textsearch	1	11
hackman	2	6
tfttp	1	2
sso	1	8

Table 4.5: Comparison *MemTrace* vs *Memcheck*

any possible memory error it detects. The higher number of reported errors makes it more difficult for the user to identify which of them may allow getting a leak. Moreover, when it detects a usage of uninitialized bytes, *Memcheck* reports only **where** the uninitialized bytes are used (i.e., the IP of the instruction), but it does not give any information about where those uninitialized bytes come from. So, the user must perform additional analysis to track down the origin of the uninitialized bytes and therefore understand whether they are interesting or not. The only case where *MemTrace* reported more items than *Memcheck* is with *cp* from *Coreutils-6.9.90*. This happens because *Memcheck* does not actually report which fields of the uninitialized parameter are accessed during the execution of system calls, and simply reports the following message:

Syscall param ⟨*syscall_name*⟩ (⟨*syscall_param*⟩) *points to uninitialized byte(s)*

Since *MemTrace* keeps track of the accesses performed by system calls and in this case some system calls access multiple uninitialized memory locations, *MemTrace* reports all of them individually, thus reporting more uninitialized reads than *Memcheck*. Furthermore, notice that *Memcheck* does not actually report the instructions performing the uninitialized reads, but it reports the instructions that use the uninitialized bytes. In some cases, a program may perform so many copies or transfers of uninitialized bytes that it is very difficult to trace backward to the instruction that first loaded the uninitialized

bytes. This is another reason why *Memcheck* reports contain so many items: the bytes read by an uninitialized read may be used in multiple locations. Indeed, while *MemTrace* only reports the uninitialized read itself (as soon as one of its uninitialized bytes is used), *Memcheck* will report an error every time one of its uninitialized bytes is used, thus possibly generating a high number of errors due to a single uninitialized read. However, it's worth noting that, as explained in [36], *Memcheck* makes use of a **bit-precision** shadow memory and that it considers as a usage of uninitialized values only some specific types of instructions. These solutions allow *Memcheck* to avoid some of the false positives that might be generated by *MemTrace*.

As pointed out by [10], *Memcheck* actually has a command-line option to keep track of the origins of uninitialized bytes. However, the concept of "origin" for *Memcheck* is different from the one used for *MemTrace*. Indeed, for *Memcheck* the origin of uninitialized bytes is represented by the instruction that allocated the memory from where the program is reading (e.g., stack allocation, call to malloc); while for *MemTrace*, the origin of uninitialized bytes is the instruction that wrote those bytes. Consider, for example, Listing 6.

```
1  int main(){
2      char* buf = (char*) malloc(sizeof(char) * 64);
3      int x, y;
4
5      x = 18;
6      y = 23;
7      *(int*)(buf + 16) = x;
8      *(int*)(buf + 20) = y;
9
10     free(buf);
11
12     unsigned long* arr = (unsigned long*) malloc(sizeof(unsigned long) *
       8);
13     unsigned long ret = arr[2]; // Loads the previously written values
      of x and y
14     printf("%p\n", ret);
15
16     return 0;
17 }
18
```

Listing 6: Multiple overlapping writes

In this small program, the values of variables x and y are written inside buffer *buf* allocated on the heap. This buffer is then freed, and an array of *unsigned long* integers is allocated so that the array has a total size equal to the size of buffer *buf*. This way, the allocator will return, for *arr*, the same address it returned for *buf*, which still contains the values of

x and y we wrote there before. Then, without initializing the elements of the array, the element at index 2 is loaded into variable *ret*, which is then printed to *stdout*. Element *arr[2]* exactly overlaps the memory locations where the values of x and y were written, as shown by Figure 4.1.

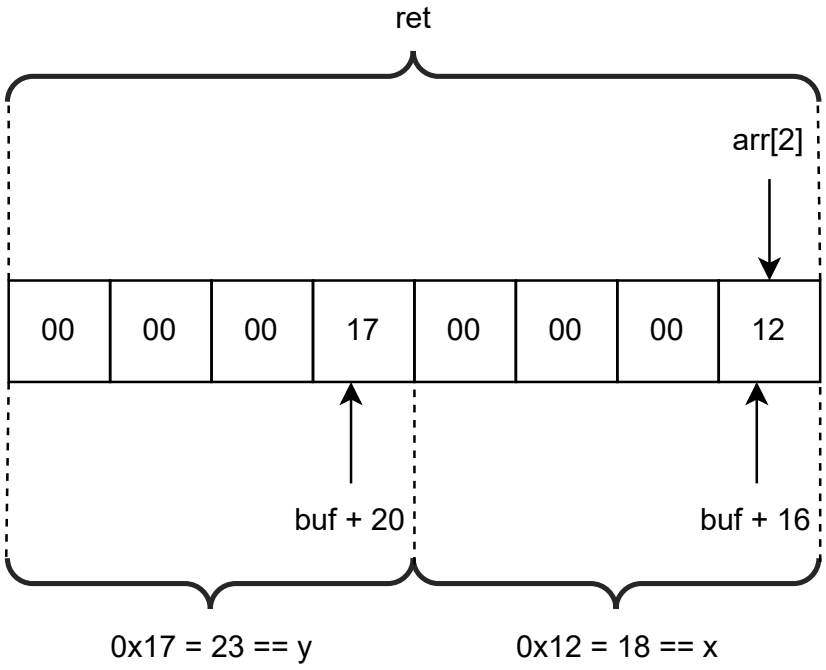


Figure 4.1: Memory layout for example in Listing 6

As the report in Listing 7 shows, *MemTrace* is able to keep track of **all** the writes that overlap the memory location read by the uninitialized read, reporting not only the address of the write accesses, but also the portion of the read memory location that they overlap as an interval of indices. *Memcheck*'s report, instead, shown in Listing 8, simply states that the uninitialized value was created by a heap allocation (i.e., a call to *malloc*), and reports the address of the instruction. While this is certainly useful for the developer which is debugging the application and is focusing on fixing the reported errors, this is not very useful if the user needs to understand what he can read leveraging the uninitialized read. Also, notice that *Memcheck* reported the uninitialized read as a conditional jump which depends on uninitialized bytes, instead of a usage of uninitialized bytes. While this is not a problem if the objective is to detect and fix errors in the code, this might be confusing if the goal is, instead, to find potential leaks.

For all the reasons listed above, we can state that despite the similarities between *MemTrace* and *Memcheck*, the tools give completely different results and act as a support for different goals. So, *MemTrace* can be considered a valuable support tool to specifically

```
LOAD ADDRESSES:
/root/test base address: 0x55de3e892000
Heap base address: 0x55de3f0a0000
/lib/x86_64-linux-gnu/libc.so.6 base address: 0x7fa015736000
Heap 1 base address: 0x7fa015a15000
/lib64/ld-linux-x86-64.so.2 base address: 0x7fa029e8c000
[vdso] base address: 0x7ffe03693000
Stack base address: 0x7ffe03649650


================================================
0x55de3f0a02b0 - 8
================================================
=> 0x11bc (0x55de3e8931bc):  W 4 B [0 ~ 3]
=> 0x11c9 (0x55de3e8931c9):  W 4 B [4 ~ 7]
   *0x11e9 (0x55de3e8931e9):  R 8 B [0 ~ 7]
================================================
================================================
```

Listing 7: *MemTrace*'s report with multiple overlapping writes

```
==13839== Conditional jump or move depends on uninitialised value(s)
==13839==    at 0x48C9C02: __vfprintf_internal (vfprintf-internal.c
   :1687)
==13839==    by 0x48B3EBE: printf (printf.c:33)
==13839==    by 0x109208: main (in /root/test)
==13839==  Uninitialised value was created by a heap allocation
==13839==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind
   /vgpreload_memcheck-amd64-linux.so)
==13839==    by 0x1091E0: main (in /root/test)
==13839==
(nil)
==13839==
==13839== HEAP SUMMARY:
==13839==     in use at exit: 64 bytes in 1 blocks
==13839==   total heap usage: 3 allocs, 2 frees, 4,224 bytes allocated
==13839==
==13839== LEAK SUMMARY:
==13839==    definitely lost: 64 bytes in 1 blocks
==13839==    indirectly lost: 0 bytes in 0 blocks
==13839==      possibly lost: 0 bytes in 0 blocks
==13839==    still reachable: 0 bytes in 0 blocks
==13839==         suppressed: 0 bytes in 0 blocks
==13839== Rerun with --leak-check=full to see details of leaked memory
==13839==
==13839== For lists of detected and suppressed errors, rerun with: -s
==13839== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Listing 8: *Memcheck*'s report with origin tracking enabled

Binary	Vulnerability detected	Argv fuzzing enabled	Notes
cbor2json	✔	✔	
md2html	✘	✔	
tail	✔	✔	
Contacts	✔	✘	
full_protection	✘	✔	
stopwatch	✔	✔	
Accel	✘	✘	Requires a custom malloc handler
Textsearch	✔	✘	Requires a custom malloc handler
Hackman	✔	✘	Requires a custom malloc handler
TFTTP	✔	✘	Requires a custom malloc handler
SSO	✘	✘	Requires a custom malloc handler

Table 4.6: Results for fuzzed execution with non-triggering inputs

help users find uninitialized reads in an executable and determine if those uninitialized reads can be exploited (e.g., to obtain a leak, execute arbitrary code..).

4.3.2. Fuzzing efficacy

The combined execution of *MemTrace* and *AFL++* successfully identified the known vulnerability for 7 out of 11 of the tested binaries. Table 4.6 reports a summary of the results. Notice that we did not perform this type of test with *cp* from *Coreutils-6.9.90*. This is because the program messes up with the files used by the fuzzer and by *MemTrace*, thus generating errors that interrupt the analysis early. Also, when argv fuzzing was not enabled, it was actually not possible to enable it, either due to the usage of a custom version of the *libc* library or because of an unusual way of starting the program, which did not use *libc*'s function *__libc_start_main*.

During the tests described in Section 4.3.1, we tested the capabilities of *MemTrace* to

Binary	Vulnerability detected	Argv fuzzing enabled	Notes
md2html	✘	✔	
full_protection	✘	✔	
Accel	✔	✘	Requires a custom malloc handler

Table 4.7: Results for fuzzed execution with triggering inputs

detect uninitialized reads by executing it as a standalone tool with a crafted input that triggered the vulnerability and we showed that *MemTrace* was able to detect it. So, the reason why the vulnerability was sometimes not detected in this configuration is probably due to the way the CFG of the program is explored. Indeed, being a dynamic analysis tool, *MemTrace* cannot report an uninitialized read if that is not executed by the program in analysis. For this reason, if the fuzzer could not generate at least 1 input that triggered the vulnerability, it could not be detected. Besides, it is worth mentioning that usually fuzzers are executed for much more time than 8 hours as they might require time to explore some paths of the CFG of the program. Therefore, it is possible that, by changing the configuration of the fuzzer (i.e., fuzzing time and/or number of parallel instances), *MemTrace* will be able to detect the vulnerability also for some of those binaries where it failed with the current configuration.

For each binary where *MemTrace* failed to detect the known vulnerability (except SSO) we also launched a new test. The setup is very similar to the previous one. However, this time we added, among the initial testcases provided to the fuzzer, a manually crafted input that was able to trigger the vulnerability and we considered the vulnerability as *detected* only if the fuzzer was able to generate other inputs that triggered it. Table 4.7 shows a summary of the obtained results. SSO has been excluded from this additional test simply because it is not possible to craft an input that triggers the vulnerability. Indeed, this binary generates a random *auth_code* for resources, and to trigger the vulnerability it requires performing a successful authentication. So, it requires executing a script that allows to leak the auth_code and use it to perform the authentication.

For *md2html*, it was expected that *MemTrace* would have not detected the vulnerability, because, as pointed out in Section 4.3.1, the uninitialized read it performs loads bytes that are used only by a *cmp* instruction.

The vulnerability in *full_protection* is a **format string error**. This kind of vulnerability

allows to leverage calls to *printf* and other similar variadic functions to either read or write arbitrary memory addresses. On the used platform, this is true as long as the memory address is higher than the stack pointer. The stack grows toward lower addresses and, in this program, there aren't local variables left uninitialized, so it is quite unlikely to read a stack location which is not initialized leveraging the format string error. During the analysis of the program performed to manually craft the input to trigger an uninitialized read, we managed to find only 1 readable stack location which was left uninitialized. The fact that the vulnerability requires a precise format string to be triggered and the low number of readable uninitialized stack locations represent a very strict condition required to detect the vulnerability, that the fuzzer was not able to satisfy. Notice, however, that format string error is not really the class of vulnerabilities that *MemTrace* is designed to detect. Indeed, format string errors are often used to get leaks by reading **initialized** values stored in well-known locations (e.g., GOT entries). So, also in this case the failure was quite expected.

With *accel*, instead, adding a triggering input as an initial testcase was enough to allow the fuzzer to generate other inputs triggering the vulnerability. So, the failure during the first test was simply due to the strictness of conditions required to be satisfied to reach the vulnerability during program's execution. Indeed, even after we added the triggering input among the initial testcases, the fuzzer only generated a few new inputs that triggered the vulnerability, and all of them generated the very same overlaps in memory. This means that there's only 1 path in the program that traverses the vulnerability, and that it is executed only when a quite strict condition is satisfied.

So, the new experimental setup detected the vulnerability only in 1 out of 3 cases. While it is not much, it is an important result, as it allows to state that *MemTrace* can also be useful in case the user already knows **where** the uninitialized read vulnerability lies, but wants to know **if** and **how** it can be controlled to get arbitrary information.

During this set of tests *MemTrace* generated a total of only 4 **false positives**. Considering that in 8 hours the fuzzer generated hundreds, if not thousands, of inputs **for each binary**, this is quite a good result, which points out that the heuristics applied by *MemTrace* are good enough to discard most of the uninitialized reads that are not due to a vulnerability. Each generated false positive has been manually analyzed to understand what it was due to. 3 of them were caused by a peculiar instance of the *stack clash mitigation* explained in Section 2.4. During the execution of *libc*'s function $\_\_stack\_chk\_fail$, which is executed when the stack canary check during a function's epilogue is failed (i.e., the canary has been overwritten), the program performs a large stack allocation. Since stack clash mitigation was enabled during compilation, the allocation follows the schema

reported by Figure 2.12. However, in this case, a probe is inserted also after the small tail allocation. Since the heuristic expects a probe to follow only allocations whose size is equal to a memory page size, *MemTrace* fails to recognize it as a false positive, and therefore reports it. The last reported false positive was due to a call to *malloc* that reads the value of the *top_ chunk*, which is considered uninitialized due to some allocations and deallocations happening before the entry point is executed. Unfortunately, there is no way to quantify the number of false negatives.

Given the length of the merged reports after 8 hours of fuzzing, it is not feasible, nor meaningful, to report all the tables they contained. Most of them are identical or very similar to the overlaps detected during the *Uninitialized reads detection tests*, which simply means that the combined execution with *AFL++* was able to detect the known vulnerability for those binaries. However, there are some interesting and more relevant overlaps which have been detected by *MemTrace* that are, therefore, reported below.

tail

Overlap 1

```
================================================
0x798a(0x7efd45e52aaa): /lib/x86_64-linux-gnu/libc.so.6 loaded at @
   0x7efd45e24000
================================================
Generated by 613 inputs:
Slave_3/id:000056,src:000000+000005,time:65382,op:splice,rep:16
Reads 8 bytes from <stack_base> - 1144
************************************************
   *0x798a (0x7efd45e52aaa): R 8 B @ (sp + 200); (bp - 5104); [0~7]
************************************************
Generated by 2053 inputs:
Slave_3/id:000056,src:000000+000005,time:65382,op:splice,rep:16
Reads 8 bytes from <stack_base> - 1144
************************************************
   0x20c9 (0x7efd45ea8b73): W 8 B @ (sp - 8); (bp - 139289784);
   [0~7]
=> 0x70b3 (0x7efd45e5cab3): W 4 B @ (sp + 136); (bp - 120); [0~3]
   *0x798a (0x7efd45e52aaa): R 8 B @ (sp + 200); (bp - 5124); [0~7]
************************************************

[...]

================================================
```

Listing 9 shows the portion of source code that causes the uninitialized read. The uninitialized read is executed by function *__mbrtowc* from *glibc*. This function declares a 64 bit integer variable called *dummy* without initializing it. The pointer of this integer value is then passed as a parameter to function *__gconv_transform_utf8_internal*, which uses it as a counter. So, as soon as this function loads its value to increase it, it performs an uninitialized read, which is detected and reported by *MemTrace*. Since *dummy* is never used again, it cannot be classified as an actual vulnerability. However, this table allows to reiterate that *MemTrace* is able to detect interesting uninitialized reads, even when they happen inside a library function, and shows that *dummy* is used, uninitialized, to fit the signature of an existing function. Being a library function, this uninitialized read will be found with every binary that calls *__mbrtowc*, including the current version of the binary (i.e., *tail* from *Coreutils-8.32*).

```
/*
###############
  From glibc/wcsmbs/mbrtowc.c
###############
*/

size_t
__mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps)
{
  wchar_t buf[1];
  struct __gconv_step_data data;
  int status;
  size_t result;
  size_t dummy; // Declared, not initialized
  const unsigned char *inbuf, *endbuf;
  unsigned char *outbuf = (unsigned char *) (pwc ?: buf);
  const struct gconv_fcts *fcts;

  [...]

  // dummy is uninitialized up to now
  status = DL_CALL_FCT (fct, (fcts->towc, &data, &inbuf, endbuf,
                              NULL, &dummy, 0, 1)); // => Calls
   __gconv_transform_utf8_internal, which uses dummy as a counter

  // dummy is never used again

  [...]

}

```

Listing 9: *glibc* uninitialized *dummy* variable

Contacts

This program implements a simple phonebook, where it is possible to store and delete contacts and set their names and an optional bio. When a contact is deleted, its content is freed, but the pointers are not overwritten with a safe value (e.g., NULL). Listing 10 shows a portion of the source code containing the vulnerability.

Overlap 1 and 2

```
================================================
0x1b3d(0x400b3d): /home/kris/Scrivania/CTFs/picoCTF/contacts/
   contacts_cpy loaded at @ 0x3ff000
================================================
Generated by 6 inputs:
Slave_0/id:000003,sig:11,src:000099,time:207901,op:havoc,rep:16
Reads 8 bytes from <heap_base> + 4200
************************************************
   0x1baf (0x400baf):  W 8 B [0 ~ 7]
   *0x1b3d (0x400b3d):  R 8 B [0 ~ 7]
************************************************

[...]

================================================
================================================




================================================
0x1ab4(0x400ab4): /home/kris/Scrivania/CTFs/picoCTF/contacts/
   contacts_cpy loaded at @ 0x3ff000
================================================
Generated by 1 inputs:
Slave_0/id:000018,sig:11,src:000044+000132,time:1043325,op:splice,
   rep:4
Reads 8 bytes from <heap_base> + 4776
************************************************
   0x1baf (0x400baf):  W 8 B [0 ~ 7]
   *0x1ab4 (0x400ab4):  R 8 B [0 ~ 7]
************************************************

[...]

================================================
================================================
```

The 2 reported tables are quite similar, as the only thing that changes is the operation that the program is performing. In the first case, the uninitialized read is executed when the program deletes a contact, creates a new one and sets the new contact's bio. In the second case, the uninitialized read is instead executed when the program deletes a contact, creates a new one and deletes this last one as well. In both the cases, the sequence of

operations leads to a **double free**, where the same heap chunk is requested to be freed twice. The double free error may allow the user to use a fake chunk to get a leak and/or to write an arbitrary address, thus allowing arbitrary code execution.

```
1  void delete_contact(struct contact *contact){
2      free(contact->name); // contact->name not overwritten
3      /* if the bio is set, free its chunk */
4      if (contact->bio != NULL){
5          free(contact->bio); // contact->bio not overwritten
6      }
7      free(contact);
8      /* replace the corresponding index with the last contact and
       decrement num_contacts */
9      for (int i = 0; i < num_contacts; i++){
10         if (contacts[i] == contact){
11             contacts[i] = contacts[num_contacts - 1];
12             num_contacts--;
13             break;
14         }
15     }
16 }
17
```

Listing 10: *contacts* source code

Textsearch

This binary from the CGC implements a sort of text search engine. Associated to each binary of the CGC, there's a README file where all the vulnerabilities of the program are listed and explained. The following overlap is caused by a vulnerability which is not listed in the README file and is, therefore, an **unintended vulnerability**.

Overlap 1

```
================================================
0x357c(0x564fb25e557c): /software/KPRCA_00036 loaded at @ 0
   x564fb25e2000
================================================
Generated by 1 inputs:
Slave_2/id:000684,src:000355+000658,time:25417114,op:splice,rep:4
Reads 1 bytes from <Heap 2_base> + 8340
************************************************
   0x30ca (0x564fb25e50ca):  W 1 B [0 ~ 0]
   *0x357c (0x564fb25e557c):  R 1 B [0 ~ 0]
************************************************
Generated by 1 inputs:
Slave_2/id:000684,src:000355+000658,time:25417114,op:splice,rep:4
Reads 1 bytes from <Heap 2_base> + 8341
************************************************
   0x30ca (0x564fb25e50ca):  W 1 B [0 ~ 0]
   *0x357c (0x564fb25e557c):  R 1 B [0 ~ 0]
************************************************

[...]

================================================
```

The uninitialized read is executed during a call to *printf*. This means that the program is printing uninitialized bytes to *stdout*. It's worth noting that in order to be able and compile CGC binaries for Linux, some library functions had to be implemented manually. In this case, the implementation of *printf*, uses *strlen* to retrieve the actual length of the string and prints one character at a time using a for loop. For this reason, all the reported reads have a size of 1 byte. This uninitialized read happens when the user asks for a text search. Under certain conditions, the program performs a double increase of a string pointer, thus skipping the string terminator and printing bytes stored in unused parts of the chunk up to the next string terminator. This vulnerability may allow a user to align heap chunks so that the call to *printf* prints interesting data such as an address, thus **obtaining a leak**.

HackMan

This binary from the CGC implements a small game where the player should guess a secret word to win. At the end of a match, the player can either quit the game or run another match. Even if the reported uninitialized reads are actually caused by the known vulnerability, in this case *MemTrace* allowed to point out a very interesting situation. The 2 tables reported by *MemTrace* are very similar, so only one of them is shown. The only difference between them was the address of the read uninitialized memory location.

Overlap 1

```
================================================
0x23ea(0x55781273e3ea): /software/KPRCA_00017 loaded at @ 0
   x55781273c000
================================================
Generated by 7 inputs:
Slave_2/id:000020,sig:11,src:000097+000322,time:16225355,op:splice,
   rep:16
Reads 8 bytes from <stack_base> - 256
************************************************
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2120); (bp + 2048); [0~0]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2121); (bp + 2049); [1~1]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2122); (bp + 2050); [2~2]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2123); (bp + 2051); [3~3]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2124); (bp + 2052); [4~4]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2125); (bp + 2053); [5~5]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2126); (bp + 2054); [6~6]
=> 0x1608 (0x7ff236b36140):  W 1 B @ (sp + 2127); (bp + 2055); [7~7]
   *0x23ea (0x55781273e3ea):  R 8 B @ (sp + 128); (bp - 96); [0~7]
************************************************

Generated by 45 inputs:
Slave_2/id:000000,time:0,orig:untriggered_input
Reads 8 bytes from <stack_base> - 256
************************************************
   0x236c (0x55f7f227e36c):  W 8 B @ (sp + 128); (bp - 96); [0~7]
   *0x23ea (0x55f7f227e3ea):  R 8 B @ (sp + 128); (bp - 96); [0~7]
************************************************

[...]

================================================
================================================
```

In the first overlap, the write accesses in the access set are performed during the execution of a *read* system call, which allows the user to insert bytes from *stdin*. The uninitialized read is caused by a call to an **uninitialized function pointer**. This means that it is possible to write inside a function pointer **arbitrary bytes** from *stdin*, thus allowing to execute arbitrary code very easily. In this case, the function pointer was called when the player decided to quit the game. There is another almost identical table (and therefore not reported), where the accessed function pointer was called when the player requested a new match. Also in that case, it was possible to fill the function pointer with arbitrary bytes from a *read* system call.

5 | Additional Tests

After the tests described in Chapter 4, we tested *MemTrace* with some other real-world binaries taken from the list in the manifest file of Ubuntu server. Unlike binaries used during tool validation, these binaries have not been compiled, but they have been installed from the standard package manager installed in Ubuntu (i.e., apt). By installing them from the package manager, the binaries were deprived of debugging symbols and they were highly optimized. Tests have been run using the combined execution with *AFL++*, using 2 parallel instances and letting the fuzzer run for 24 hours on the same remote machine used for validation tests.

We tested 130 binaries from the Ubuntu server manifest. While no vulnerabilities have been found, these additional tests allowed to state the feasibility of the analysis with complex, real-world binaries (e.g., compilers and linkers); to verify the goodness of the heuristics applied by *MemTrace* and, finally, to classify the most frequent sources of false positives generated by *MemTrace*. With most of the tested binaries, very few false positives were generated (1 - 5), and the one that generated more false positives has been *x86-64-linux-gnu-gold*, a linker from package *binutils*. After 24 hours of fuzzing, the merged report contained only 31 tables. While this number is much higher than the number of tables reported in Chapter 4, it's worth noting that the fuzzer ran for much more time and that this binary is much more complex, as it contains many branches and possible paths. Indeed, the fuzzer generated, for this binary, more than 2,000 inputs for each fuzzer instance, thus generating more than 4,000 inputs in total. Given the number of inputs analyzed by *MemTrace*, the number of generated false positives is more than reasonably low. Moreover, many of the false positives reported for a binary are usually very similar as they present the same pattern, thus making them easily and quickly recognizable.

By analyzing the reports generated by *MemTrace*, we noticed that the main sources of false positives are string function optimizations, stack clash mitigation applications, alignments of fields in *structs*, usages of *realloc* and bit-fields. We already discussed string functions optimizations and stack clash mitigation in their respective paragraphs in Section 2.4. Therefore, the first source of false positives we're going to discuss about are alignments

of fields in structs. The fields of a struct are allocated sequentially in the same order they are declared in, adjacent one to the other. So, a *struct* size is usually equal to the sum of the sizes of the data types it is composed of. However, sometimes the compiler allocates more space than that. This is done simply for alignment purposes. Indeed, unaligned memory accesses are usually slower than aligned accesses, as the processor might require to load from memory more than 1 word (i.e., the unit of data managed by the processor). For this reason, the compiler may add some gaps between adjacent struct fields in order to align all of them according to their types and therefore optimize memory accesses. While this is usually not a problem, as it is quite unlikely that the program uses bytes from the gaps inside structs, sometimes some optimizations may cause accesses to those bytes. For instance, if a string function is called passing as a parameter a string which is adjacent to a structure, the applied optimizations may load bytes belonging to the struct, which may include bytes used for padding, which are, of course, in an uninitialized state. Moreover, sometimes also structs are copied from a memory location to another one. The compiler usually optimizes these copies by using *memset*, instead of copying each field one by one. By doing this, however, also the padding bytes are copied, thus propagating their uninitialized state into other memory locations, and therefore increasing the chances of generating a false positive.

Another main source of false positives is the usage of function *realloc* and similar. This type of function takes as parameters the address of an allocated heap chunk and a size, and it will return a new heap chunk whose size is at least equal to the given one. To do this, the allocator checks if it can simply allocate more space to the passed chunk, in which case the same address is returned; while if it is not possible to increase its size (i.e., there are other adjacent allocated chunks) a new chunk is allocated, and the content of the old chunk is copied in the new one, including, if any, the bytes that were in an uninitialized state. By copying these uninitialized bytes, the *realloc* function is performing an uninitialized read, and the uninitialized state is therefore propagated in the new memory location. As it happens with struct fields alignment, *MemTrace*'s taint analysis discards most of these uninitialized reads, as most of the uninitialized bytes will never be used by any instruction. However, it is possible that some optimizations actually access those bytes, thus causing *MemTrace* to report the uninitialized read performed by *realloc*, and therefore generating a false positive. During tests, this usually happened when the *realloc* function was used to implement a dynamic array of characters to allow the program to accept a string of unknown length from *stdin* or from a file. Being a string, it is usually managed using string functions and, as soon as their optimizations use some of the uninitialized bytes copied by *realloc*, *MemTrace* reports the uninitialized

read it performed while copying the content of the old array.

The last frequent sources of false positives are bit-fields. According to [15]:

> *(A bit-field) declares a class data member with explicit size, in bits. Adjacent bit-field members may (or may not) be packed to share and straddle the individual bytes.*

So, bit-fields are used when it is necessary to declare inside a struct some fields which have a size lower than 1 byte (e.g., a bunch of boolean flags). However, memory is accessed with a byte granularity. This means that whenever the CPU performs a load from memory, it must load at least 1 byte. The usage of bit-fields in the source code is automatically managed by the compiler, which converts assignments and loads of bit-fields values in a sequence of bitwise operations. Let us consider, as an example, the structure defined in Listing 11.

```
struct st{
    int n;
    char c;
    unsigned b1 : 1;    // Bit-fields grouped together in
    unsigned b2 : 1;    // the same byte
    unsigned b3 : 1;    //
}
```

Listing 11: Bit-fields example

Fields b1, b2 and b3 are bit-fields of size 1 bit. The compiler will therefore group them together in the same byte and they will be stored starting from the least significant bit. Let us assume the program assigns a value 1 to field b1. In order to change the value of the corresponding bit, also keeping every other bit untouched, the compiler converts this assignment in the following sequence of pseudo-instructions (or something equivalent):

```
LOAD al, st.b1;
OR al, 1
STORE st.b1, al
```

Since the 5 most significant bits of the byte containing b1 are unused, the LOAD instruction will perform an uninitialized read, and the execution of the bitwise OR will use those uninitialized bytes. So, *MemTrace* will report the LOAD instruction as an uninitialized read, thus generating a false positive.

6 | Limitations

6.1. Prototype limitations

Currently, *MemTrace* has been implemented and tested on a 64 bits x86 architecture only. In order to be used on 32 bits x86 architecture, the tool requires some modifications. However, since **generality** and **extensibility** have been the main concerns during the whole design and implementation phases, it is enough to change or add the implementation of the platform-specific components of *MemTrace*, thus reducing to a minimum the amount of changes required in the existing source code. The same holds also for the underlying operating system. Indeed, *MemTrace* has been implemented and tested on machines running Ubuntu with *glibc-2.31* installed. So, the tool should work without requiring any change on each Linux distribution using the same version of *glibc*. It may work also if another version of *glibc* is used, as long as the layout of heap chunks is the same as *glibc-2.31*. Once again, if needed, it is sufficient to change or add the implementation of the platform-specific components to port *MemTrace* to other OSs. Of course, the target platform must be supported by *Intel PIN* in order to allow porting *MemTrace* (e.g., 32 or 64 bits x86 are the only architectures supported by *Intel PIN*).

Also, *MemTrace* currently supports only single-process, single-threaded applications. It is certainly possible to extend it to support multi-threading, but this will require some heavy changes, as it requires at least to synchronize accesses to some global variables during the analysis of program's threads.

6.2. Inherent and inherited limitations

MemTrace is implemented as a dynamic analysis tool, which means it must execute the program to be able to analyze it. This leads to an inherent limitation. Indeed, *MemTrace* can only analyze those instructions, and therefore those memory accesses, that actually happen during program's execution. So, if a certain path in the program CFG is never executed and contains an uninitialized read, *MemTrace* will not be able to detect and report it, thus generating a false negative, which is difficult to be tracked. As discussed

in Chapter 3, we tried to mitigate this limitation by combining *MemTrace* with a fuzzer. We use AFL++, which is designed to generate inputs that explore new paths of the program. This way, we can greatly increase the path coverage, but it can still not be enough. Indeed, even fuzzers themselves have inherent limitations. For instance, they might fail to generate an input able to traverse a certain path. This may happen when a path is executed only if a very strict condition is verified. Fuzzers, in fact, generate new inputs by getting some random bytes and/or combining bytes from other inputs. So, if a condition is satisfied only with very few values of some variables, the inputs generated by the fuzzer might not be able to satisfy it. In those cases, of course, *MemTrace* will not be able to analyze that path. Moreover, fuzzers require a great amount of time to explore large parts of the CFG, and the more complex a program is, the more time it might need.

Another existing limitation, that affects the tracing of memory accesses performed on the heap, is due to the usage of *Intel PIN* as the underlying DBI framework. Being implemented as a dynamic analysis tool, *MemTrace* can be used even if the executable has been *stripped*, i.e., it has been deprived of all the symbols information about functions and variables. However, in order to be able to detect the usage of *malloc, free* and similar dynamic memory management functions, it is required to have the information about the symbols of these functions available. Indeed, in order to instrument a specific function, *Intel PIN* requires to build a corresponding RTN object, which is the way Intel PIN internally models routines (i.e., functions). To do this, it allows to lookup for a specific function **by name** inside the loaded images (e.g., executable or shared libraries), but this lookup is actually successful only if the image containing the searched function has their symbols information available. Usually, these functions are implemented in a system-wide installed shared library and its symbols information is usually available to allow easy debugging of applications, or it is anyway possible to easily retrieve them. But if this is not the case, it might prevent *MemTrace* from correctly keeping track of memory allocations and usages on the heap.

6.3. Implementation limitations

Some of the design and implementation choices come with some drawbacks. Consider the *system call manager* we discussed in Section 2.4. It certainly helps tracing the memory accesses performed during the execution of a system call, but it requires implementing the specific handler for that system call. We implemented the handlers for many system calls for 64 bits x86 architecture, but we did not implement a handler for all of them. Indeed, some system calls do not perform any memory access, so it would be useless to implement

a handler for them. For what concerns system calls performing memory accesses, we implemented the handlers for most of them, for a total of 116 implemented handlers. However, there might be some more rarely used system calls not having a corresponding handler, thus possibly generating false negatives, if the program uses them. Moreover, system call handlers are sometimes just an approximation of the actual behavior of a system call. This is usually the case when a system call has a very complex behavior where a data structure is only partially accessed according to the system call parameters. In these cases, the handler does not usually consider each possible parameter, but simply considers the data structure as completely accessed. By not modeling that behavior in an exact manner, we make the handler much simpler and faster, thus avoiding slowing down the analysis too much. Notice, however, that these approximations do not happen frequently and that they always allow to detect the uninitialized reads performed during the execution of the corresponding system call, although they possibly generate a few false positives. Anyway, these false positives can be easily discarded by analyzing the textual report and verifying the system call behavior with a debugger.

A quite similar problem is caused by the *instruction manager* described in Section 2.4. As mentioned, we implemented some default handlers valid for most of the instructions, but some of them required a specific handler to be implemented. Of course, it is not feasible to implement a handler for each instruction available in the ISA, so we had to choose which instruction handlers were worth implementing. We implemented the handler for some of the most commonly used instructions, so that we could cover as many cases as possible, but all the others don't have a specific handler. While this is easily fixable by implementing the missing handlers, it must be considered as a limitation of *MemTrace*, as it might prevent it from working well, if not carefully addressed.

Finally, despite it was not the main concern of our approach, another limitation is represented by the command-line arguments fuzzing. As explained in Chapter 3, it is performed by using some of the bytes generated by the fuzzer as arguments for the binary. While it works well enough to allow *MemTrace* to explore more paths of the program being analyzed, it is certainly not optimal, as it might not be able to generate each argument accepted by the program or generate combinations of arguments that trigger some particular behavior, thus preventing *MemTrace* from exploring the paths they allow to traverse. For instance, during testing, we noticed that the used approach for command-line arguments fuzzing is able to generate only those arguments whose length is quite limited (i.e., up to 3-4 characters). However, often programs accept a set of long arguments whose size is much longer than 4 characters, and that are often composed of more words concatenated by a hyphen. Given the limited capability of the used approach to generate

long valid command-line arguments, it is possible that many of the existing paths of an executable are never executed by *MemTrace*.

7 | Future work

The results depicted in Chapter 4 show *MemTrace*'s effectiveness in detecting uninitialized read accesses in a program. However, there's certainly room for improvements. As a first thing, it might be possible to further improve the heuristics used to detect the false positives due to optimized string functions or to stack clash mitigation. Indeed, they are among the main sources of false positives, as they are still unable to detect and ignore all of them and might even cause false negatives. By further focusing on those heuristics and the cases they must handle, it may be possible to improve them, thus further reducing the number of reported false positives and/or of false negatives. It may also be possible to apply the string function optimizations heuristic even in the case of indirect uninitialized reads. This way, the number of false positives generated due to the copies performed by *realloc* should drop, but it may even cause a great increase in the overall analysis time. So, this requires more testing and analysis to verify whether the results obtained by this approach are worth the increase in the execution time.

For what concerns the false positives generated due to struct padding and bit-fields, instead, it would be required to discover the layout and boundaries of memory objects. This is not an easy task, and, indeed, it is still an open research topic ([8, 13, 38]) and therefore out of the scope of *MemTrace*. However, it might be worth trying to apply some of the existing approaches or to use some of the existing tools to try and reduce the number of false positives, but it of course requires many further tests and an accurate analysis to check that the execution time does not increase too much, thus possibly making the analysis infeasible with real-world binaries. Alternatively, it might be possible to try and deal with these types of false positives by switching to a *bit-precision* shadow memory and by handling more precisely bitwise operation instructions, so that not all of them will be considered as a usage of uninitialized bytes, in a manner similar to what it is done in [31].

Then, we will deal with *MemTrace*'s current limitations, trying to solve or at least mitigate them. For what concerns system call and instruction handlers, we will perform a more in-depth analysis of the most frequently used system calls and instructions, so that we can implement the handlers that are currently missing, and might therefore prevent *MemTrace*

from working properly. Afterwards, *MemTrace* will of course be extended to support 32 bits x86 architecture and multi-threaded applications in order to broaden its applicability.

After that, the most challenging limitation to deal with is coverage. First of all, we will try to improve the command-line arguments fuzzing. Indeed, this improvement would allow generating more arguments accepted by the program, thus possibly allowing *MemTrace* to explore new paths. Usually, in order to check if a certain argument has been passed by the user, programs compare the values from *argv* with some literal strings containing the argument itself. Furthermore, they usually declare a string literal containing the usage manual of the program itself, which in turns lists and describes every accepted argument. String literals are hardcoded into the binary, usually in a read-only section. So, a possibility to improve the arguments fuzzing could be using an external tool (e.g., *strings* from binutils) to extract those hardcoded string literals and combine them with the bytes generated by the fuzzer to generate the arguments for the program under analysis.

While fuzzing certainly allows increasing coverage, it requires a lot of time to explore the CFG of a program, also according to its size and complexity. It would be better if the exploration of the CFG could be faster and more controllable, instead of being completely dependent on the random and genetic algorithms usually applied by the fuzzer. A possibility could be using a symbolic execution engine to combine fuzzing with a **concolic execution**, similarly to what is done by Driller [2, 40]. This way, while the fuzzer quickly generates inputs to explore paths with loose conditions, the symbolic execution engine may use a real execution context as initial state, and start a symbolic execution from there, building constraints to execute those paths with more strict conditions and solving them to find some feasible values. Of course, to make this approach efficient, it is necessary to keep track of which paths have been explored, so that symbolic execution can be interrupted as soon as it starts to execute an already explored path in order to avoid path explosion. For this reason, this approach may also need to perform a preliminary static analysis to retrieve the CFG of the program. Notice, however, that the CFG will not be used to perform the analysis, but will be used only to detect branches and keep track of which of them have already been explored. So, the quality of the retrieved CFG will not affect the analysis itself, but only path exploration, that, in the worst case, would be the same as the currently achieved level. Furthermore, the fuzzer should be able to leverage the inputs generated by the symbolic execution engine in order to generate, in turn, new inputs. This is certainly not a straightforward solution, but it might help explore more paths in a smaller time span.

8 | Conclusions

The main goal of our work was to design an analysis tool to detect uninitialized reads in a binary executable that might allow leaking information. For this purpose, we developed *MemTrace*, which makes use of *Dynamic Binary Instrumentation* to keep track of the memory accesses performed by a program and generates a report containing all the memory overlaps, that is all the uninitialized reads grouped with all the write accesses that overlap the same memory location. In order to let the tool explore paths in a program and discover potential vulnerabilities, we also paired *MemTrace* with *AFL++*.

We tested our tool setting up different types of tests. First, we tested the execution time overhead introduced by *MemTrace* and compared it with the overhead introduced by *Memcheck*, which is implemented in a similar way, showing that *MemTrace*'s overhead is reasonably higher, given the additional amount of operations performed for each memory access. Then we verified the capability of *MemTrace* to detect uninitialized read vulnerabilities. To do so, we launched *MemTrace*'s analysis on several binaries with a known uninitialized read vulnerability and verified that the reports generated by *MemTrace* pointed out that vulnerability, also verifying the presence and the causes of false positives. Finally, we verified the efficacy of the combined execution with *AFL++*. We used the same binaries used for the previous tests, and let the fuzzer run for 8 hours using random inputs and inputs not triggering the vulnerability as initial testcases. When the analysis was completed, we checked if it detected the known vulnerability, finding out that the vulnerability was found for most of the binaries. Each report has been analyzed manually, also verifying the number and the causes of false positives. Besides detecting the known vulnerabilities, *MemTrace* helped to detect a usage of a dummy uninitialized variable in a function from *glibc* and an unintended vulnerability in a CGC binary which may allow to leak information. Moreover, *MemTrace* helped detect some interesting situations that might occur due to some of the known vulnerabilities, as it pointed out the existence of a double free vulnerability in a binary from a CTF and the possibility to fill function pointers from *stdin* in another CGC binary.

After tests done for validation, we also performed some additional tests that allowed us

to classify the most frequent sources of false positives and to explain why they generate an uninitialized read.

Finally, we also analyzed the limitations that affect *MemTrace*, dividing those due to the used approach from the ones due to the current implementation or inherited by the used frameworks and external tools. Among them, the main and most challenging limitation of *MemTrace* is coverage. Indeed, the usage of a fuzzer is not enough to always explore all the branches of a program's CFG, and, besides, it may require a large amount of time to do the exploration.

The results obtained during testing prove that *MemTrace* is a valuable tool to support the user detecting potential leaks in a binary and they also show that *MemTrace* is able to help detect even other types of vulnerabilities related to uninitialized reads.

Despite being an immature tool which might be improved in future, *MemTrace* can be considered as another step toward automatic exploitation. Indeed, it performs an automatic analysis of binary executables looking for potentially exploitable uninitialized read vulnerabilities. Out there, other tools exist that are able to perform an automatic analysis of executables looking for other types of vulnerabilities, and even automatically generate an exploit for them ([6, 12]). However, they usually do not deal with enabled mitigations such as ASLR and stack canaries. By combining *MemTrace* with these tools, it might be possible to analyze a binary and leverage the information reported by *MemTrace* to generate an exploit to execute arbitrary code also bypassing the enabled mitigations.

Bibliography

[1] Aflplusplus. URL https://github.com/AFLplusplus/AFLplusplus. (Last visit: 15/03/2022).

[2] Driller. URL https://github.com/shellphish/driller. (Last visit: 15/03/2022).

[3] Cyber grand challenge, 2014. URL https://www.darpa.mil/program/cyber-grand-challenge. (Last visit: 15/03/2022).

[4] Bringing stack clash protection to clang / x86 — the open source way, 2021. URL https://blog.llvm.org/posts/2021-01-05-stack-clash-protection/. (Last visit: 15/03/2022).

[5] *SimProcedure inaccuracy*. angr. URL https://docs.angr.io/advanced-topics/gotchas#simprocedure-inaccuracy. (Last visited: 18/03/2022).

[6] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. 2011. URL https://dl.acm.org/doi/abs/10.1145/2560217.2560219.

[7] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. 2010. URL https://ieeexplore.ieee.org/document/5427374.

[8] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of the Eighth International Conference on Verification, Model Checking and Abstract Interpretation*. Springer International Publishing, 2007. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1019&rep=rep1&type=pdf.

[9] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao.

[10] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *Pro-*

ceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), 2007. URL https://valgrind.org/docs/origin-tracking2007.pdf.

[11] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011. URL https://ieeexplore.ieee.org/document/5764689.

[12] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012. URL https://ieeexplore.ieee.org/document/6234425.

[13] K. Chen, Y. Zhang, and P. Liu. Dynamically discovering likely memory layout to perform accurate fuzzing. 2013. URL https://ieeexplore.ieee.org/document/7386711.

[14] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. 2009. URL https://llvm.org/pubs/2009-06-HotDep-SymbolicExec.pdf.

[15] *Bit-fields*. cppreference.com. URL https://en.cppreference.com/w/cpp/language/bit_field. (Last visit: 15/03/2022).

[16] G. J. Duck and R. H. C. Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. 2018. URL https://dl.acm.org/doi/10.1145/3296979.3192388.

[17] eliben. pyelftools. URL https://github.com/eliben/pyelftools. (Last visit: 18/03/2022).

[18] A. D. Federico, M. Payer, and G. Agosta. Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries. In *CC2017*, 2017. URL https://rev.ng/downloads/cc-2017-paper.pdf.

[19] A. Fioraldi, D. C. D'Elia, and L. Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, 2020. URL http://www.diag.uniroma1.it/~delia/papers/secdev20.pdf.

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

[21] B. Garmany, M. Stoffel, R. Gawlik, and T. Holz. Static detection of uninitialized

stack variables in binary code. In *European Symposium on Research in Computer Security*, 2019. URL https://arxiv.org/abs/2007.02314.

[22] D. Gopan, E. Driscoll, D. Nguyen†, D. Naydich†, A. Loginov, and D. Melski. Data-delineation in software binaries and its application to buffer-overrun discovery. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015. URL https://ieeexplore.ieee.org/document/7194569.

[23] C. Hauser, J. Menon, Y. Shoshitaishvili, R. Wang, G. Vigna, and C. Kruegel. Sleak: automating address space layout derandomization. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*, 2019. URL https://dl.acm.org/doi/abs/10.1145/3359789.3359820.

[24] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/heelan.

[25] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019. URL https://dl.acm.org/doi/10.1145/3319535.3354224.

[26] J. Hu, J. Chen, L. Zhang, Y. Liu, Q. Bao, H. Ackah-Arthur, and C. Zhang. A memory-related vulnerability detection approach based on vulnerability features. 2020. URL https://ieeexplore.ieee.org/document/9036137.

[27] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel. URL https://cdrdv2.intel.com/v1/dl/getContent/671200. (Last visit: 15/03/2022).

[28] E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011. URL https://dl.acm.org/doi/10.1145/2024569.2024571.

[29] G. Lettieri. Stack canaries, 2020. URL https://lettieri.iet.unipi.it/hacking/canaries.pdf. (Last visit: 15/03/2022).

[30] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. 40, 2005. URL https://dl.acm.org/doi/10.1145/1064978.1065034.

[31] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, June 2007. URL `https://valgrind.org/docs/shadow-memory2007.pdf`.

[32] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, 2007. URL `https://valgrind.org/docs/valgrind2007.pdf`.

[33] M. Neugschwandtner, I. Haller, P. Comparetti, and H. Bos. The borg: Nanoprobing binaries for buffer overreads. In *CODASPY '15: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015. URL `https://dl.acm.org/doi/abs/10.1145/2699026.2699098`.

[34] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. 15, 2012. URL `https://dl.acm.org/doi/10.1145/2133375.2133377`.

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012. URL `https://dl.acm.org/doi/10.5555/2342821.2342849`.

[36] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, 2005. URL `https://valgrind.org/docs/memcheck2005.pdf`.

[37] E. C. Sezer, P. Ning, C. Kil, and J. Xu. Memsherlock: An automated debugger for unknown memory corruption vulnerabilities. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, 2007. URL `https://www.researchgate.net/publication/221609155_Memsherlock_an_automated_debugger_for_unknown_memory_corruption_vulnerabilities`.

[38] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *The Network and Distributed System Security (NDSS) Symposium 2011*, 2011. URL `https://www.cs.vu.nl/~herbertb/papers/howard_ndss11.pdf`.

[39] D. Song, D. Brumley, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *In Proceedings of the 4th International Conference on Informa-*

tion Systems Security, 2008. URL http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf.

[40] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium (NDSS) Symposium*, 2017. URL https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[41] D. Upton, K. Hazelwood, R. Cohn, and G. Lueck. Improving instrumentation speed via buffering. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*, 2009. URL https://dl.acm.org/doi/abs/10.1145/1791194.1791202.

[42] *Memcheck: a memory error detector*. Valgrind, 2007. URL https://valgrind.org/docs/manual/mc-manual.html. (Last visit: 15/03/2022).

[43] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. Qsym: a practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018. URL https://dl.acm.org/doi/10.5555/3277203.3277260.

Acronyms

DBI *Dynamic Binary Instrumentation*

CFG *Control Flow Graph*

CGC *Cyber Grand Challenge*

CTF *Capture The Flag*

List of Figures

List of Tables

List of Listings

List of Algorithms

Acknowledgements

Ringrazio innanzitutto i professori M. Polino e M. Carminati, i quali mi hanno guidato e seguito durante tutte le fasi di questo progetto che rappresenta la conclusione di un lungo ed importante percorso.
Ringrazio inoltre i colleghi con i quali ho instaurato un rapporto di amicizia, che sono sempre stati aperti al confronto, intellettuale o dilettevole che fosse, e che mi hanno perciò spronato a proseguire facendo sempre del mio meglio. È stato un piacere ed un onore aver condiviso parte del percorso con voi.
Ringrazio anche tutti gli amici e i parenti, che sono sempre stati un insostituibile supporto e che nei momenti più difficili avevano sempre una parola di conforto o di incoraggiamento per me.

Ma soprattutto, vorrei fare un ringraziamento speciale ai miei genitori, i quali hanno fatto enormi sacrifici per fare in modo che arrivassi dove sono oggi.
Senza di loro non sarei la persona che sono. Sono le mie colonne portanti: mi hanno sempre supportato e sostenuto in tutto ciò che ho fatto e anche quando dubitavo delle mie stesse capacità, c'erano loro a credere in me, e questo era sufficiente a darmi la forza di provare.
Sono sempre stati pronti a festeggiare insieme a me i successi e a darmi una mano per affrontare le sconfitte.
Per questo motivo, tutto ciò che sono lo devo anche e soprattutto a loro.
Grazie, Mamma e Papà.