Detection of unbounded memory accesses using dynamic taint analysis

Master's thesis

Zhani Baramidze

Supervisors: Prof. Dr.-Ing. Wolfgang Kunz & M.Sc M. Ammar Ben Khadra

Department of electrical and computer engineering
University of Kaiserslautern

September 20, 2018

Declaration of Authorship

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Student's signature: _____

Name (in capitals): _____

Date of submission: _____

Abstract

Exploitation of memory errors has been going on for more than 25 years already. A lot of tools have been developed with both defensive and offensive security capabilities but none of them has been able to offer the perfect solution by far. Couple years ago, in 2014, buffer over-read vulnerability was detected in OpenSSL, a widely used software library containing an open-source implementation of the SSL and TLS protocols. This left half a million widely trusted websites vulnerable [19].

In this thesis, we have built a memory error checking tool KATU. KATU tries to detect only a specific family of memory errors, which would allow an attacker to access unbounded memory. Heartbleed is one example of such a bug. To be able to detect the vulnerability, our tool doesn't need the out-of-bounds memory access to happen in the application. Instead, it's enough if the application follows the control-flow path on which the out-of-bounds memory access would happen for correct inputs to the program. To keep track of and propagate information about the bounds, we have used Dynamic Taint Analysis technique. The tool keeps track of applied restrictions and thus boundedness information of every tainted location and thus is able to tell if the tainted memory address is bounded.

We have used DynamoRIO, an open source dynamic instrumentation tool platform and developed our tool as an extension of DynamoRIO.

Contents

Chapter 1

Introduction

Memory safety is the key to program's correctness. If a program is not memory safe, there is a possibility of memory corruption that can lead to incorrect behavior and can even be exploited. So, what is memory safety, and why is it so important? Amorim et al., 2018 [9] tries to give a formal characterization of what it means for a programming language to be memory safe, while other papers give more informal, straightforward definition, stating that memory safety is a characteristic of a program or a programming language, which guarantees that memory access errors never occur. Memory access errors can be split into two different kinds: spatial and temporal errors. Spacial memory errors are violations caused by dereferencing pointer that is out of bounds. This can be caused by indexing array outside bounds, more generally invalid pointer arithmetic, or by dereferencing uninitialized or NULL pointers. Temporal errors, on the other hand, are violations caused by dereferencing pointer that is not valid anymore, because it was freed (dangling pointers). Freeing already deallocated memory is also a temporal violation. The reason why memory safety is so important is that its violations lead to undefined behavior, that can often be exploited. To illustrate how those vulnerabilities can be exploited, we give the following example:

In this example, we have a spatial memory vulnerability, or more concretely a buffer overflow vulnerability. To see how it can be exploited we should first take a look at a memory layout when `func1` gets called at Figure 1.1. Note that we have ignored possible padding that compiler can introduce here as well as additional variables for various gcc features (e.g. guard variables for stack protection).

In `func1`, if *scanf* reads more then 4 bytes, it will overwrite `is_admin` variable, which can lead to gaining some privileges. Apart from that, if *scanf* reads even more characters it will first overwrite the old program counter value, and then return address. This way, an attacker can place machine instructions on the stack and then make function jump to (by rewriting return address) that code, making it possible to execute arbitrary code.

From this simple example, it's clear that memory safety is crucial for se-

```
struct my_struct
{
  char buffer[4];
  int is_admin;
};

void func1()
{
  struct my_struct s;
  s.is_admin = check_privileges();
  scanf("%s", s.buffer);
}
```

Listing 1: Example of stack overflow vulnerability.

curity reasons. Some high-level languages like Java are memory safe. Java achieves memory safety by doing various runtime checks when indexing array and dereferencing pointers. Such checks guarantee that whenever a spacially unsafe operation is attempted in Java applications, they get detected and the application gets notified. Temporal memory safety, on the other hand, is addressed by garbage collectors and reference counting. However, those checks come with a performance penalty and are not always affordable.

1.1 History

Due to the necessity, a lot of different kind of techniques are getting invented to overcome memory safety issues in applications compiled from memory-unsafe languages such as C and C++. It all started in 1988 when the Morris Worm incident happened on the internet and the buffer overflow attack gained notoriety [31]. After that, there have been numerous attempts to overcome this issue. StackGuard [8] was one of the first widely adopted tools that tried to solve the problem. It got implemented in 1997 as a set of patches to GCC 2.7 and was published at the 1998 USENIX Security Symposium. StackGuard works by placing a random word, called Canary word, before return address on the stack and saves this random value. Before function returns, it compares placed canary word with its original value to see if it has changed during function execution. If it did, it would mean that somewhere on the stack buffer overflow had happened and return address could have been overwritten. This simple yet useful feature was later integrated into GCC under name “stack-protector”.

Canaries, however, are not impossible to overcome. First of all, an attacker can figure out the value of the Canary word by exploiting some other vulnerability. Apart from that, So-called Structured Exception Handler (SEP)

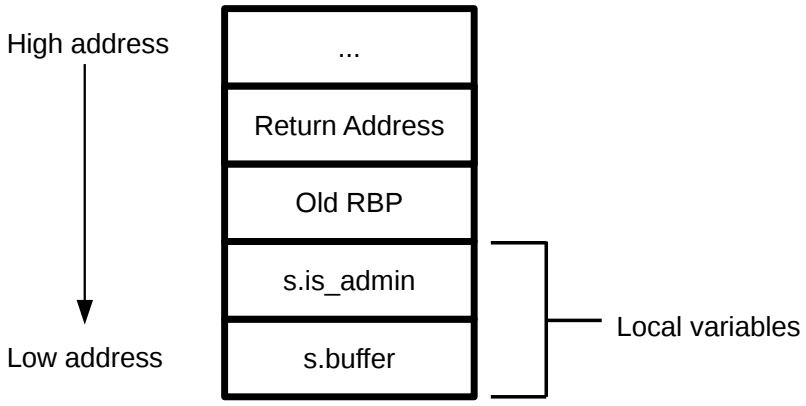


Figure 1.1: Memory layout.

Exploitation can be used to overcome Canary protection [13]. And lastly, it only protects against stack overflows, not against other kinds of memory safety vulnerabilities.

Another useful and widely deployed defense mechanism against memory safety vulnerabilities is so-called "Executable space protection". This mechanism marks regions of memory as non-executable, making execution of injected instructions impossible. In most cases, this is achieved by a hardware feature called "NX (no-execute) bit", in page table entries. Support for this feature has been part of Linux kernel mainline since kernel 2.6.8. When kernel loads the executable, it checks for ELF headers, determines which memory pages will contain code and grants execute permission only to those pages. Typically these sections are .init/.fini (containing initialization/termination codes), .plt/.plt.got (used to jump to shared libraries) and .text (main codes of the program). Windows calls it "Data Execution Prevention" (DEP) and was first implemented in Windows XP Service Pack 2 (2004) and Windows Server 2003 Service Pack 1 (2005).

NX-bit, however, doesn't protect from modifying return address to jump to another, already existent code of attacker's choice instead of injecting. For example, since most of the applications written in C have the standard C library (libc) linked, an attacker could jump to one of the routines provided by libc. This is called the return-to-libc attack. Since on x86 architectures mostly *cdecl* calling convention is used, an attacker can very easily customize arguments that she wants to pass to libc function by placing them on the stack. This gets complicated on 64-bit machines, where the first couple arguments are placed in registers instead.

In 2007, Shacham pesented a paper where he showed a technique that allowed

a return-to-libc attack to be mounted on x86 that would call no functions at all [28]. Later, this technique was generalized into a so-called "Return-Oriented Programming (ROP)". By this technique, malicious computation is achieved by linking together short code snippets already present in the program's address space. Those snippets are called "gadgets" and all of them end with 'ret' instruction. Those gadgets can even be generated automatically [26]. Since attacker controls the stack, it arranges things on the stack so that first gadget is executed first, which then jumps (returns) to the second gadget and so on. Since all the gadgets are already in code memory, NX-bit cannot help with this kind of attacks. An attacker can then, for example, call mprotect()/VirtualProtect() functions on Unix/Windows, correspondingly, which can make stack executable. After making stack executable attacker can just inject any code in the stack and execute it.

One commonly deployed defense mechanism that makes ROP harder is Address space layout randomization (ASLR). ASLR randomly arranges memory regions of the process, like text, stack and heap. This makes harder for an attacker to jump to a particular function or gadget because it is randomly arranged. Linux kernel 2.6.12 enabled a weak version of ASLR by default. Microsoft enabled it first in January 2007, in Windows Vista.

Some of the widely used ASLR implementations have weaknesses. For example, on 32-bit machines, since address space is not too large (4GB) and page alignment must still be maintained (4KB), brute-force attacks are possible due to low entropy [18]. Apart from implementation-related weaknesses, ASLR's biggest threat is information leakage [32], which means exploiting memory corruption vulnerability to obtain current code address, which can then be used to calculate the random offset applied by ASLR.

1.2 Approaches

As we have discussed, currently deployed defense mechanisms cannot fully protect against exploiting memory safety vulnerabilities. Thus, we need additional tools to analyze our software for such potential vulnerabilities.

Some of those tools are static, which means they analyze software without running it, only looking at its instructions or source code. Thus, they reason about all possible executions of the code. This kind of analysis has an advantage that it doesn't produce any runtime overhead since it's only done offline. However, due to the undecidability of static analysis in general [17], static analysis tools have to rely on overapproximation that can lead to false positives. There are some static analyzers for programs written in the C language, but they impose some restrictions on C code. For example, The ASTREE analyzer [7] is such a tool that supports simplified version of C. Code that ASTREE works on should not have malloc or recursion, for example. Plus, static analysis tools tend to produce a lot of false positives [33]. Such analysis can, on the other hand, help eliminate unnecessary checks (e.g. checks that are made redundant by previous checks) and thus improve performance. Compilers have some sort

of static analysis tools inside, which can generate warnings when compiling vulnerable code or optimize redundant checks out. Another usage of static analysis is in control/data flow integrity tools. Those tools compute control/data flow graphs using static analysis and add instrumentation code to an application that ensures that either the jumps are made to/from correct locations (control flow graph) or data flow at any moment is allowed (data flow graph).

Other tools are dynamic, which means they do the analysis on runtime. GCC and Clang refer to such tools as "Sanitizers" and they analyze only a single, given execution of the program. We will shortly analyze some of the most common techniques that those tools use to detect bugs [30]:

1. **Red-zone Insertion**: Some of the tools insert so-called "red-zones" between allocated memories. This way, when the program will try to dereference a pointer that is out of bounds, it will try accessing red zone, which the tool will notice and write an appropriate warning. In order to detect access to red-zone, tools have to monitor every memory access that the software makes, which can be done by either modifying operating system kernel to call some hook on every memory access, or modifying program's object code itself to call the hook before every memory access. Apart from that, the tool will have to keep information about the red zones and their locations. This is usually achieved by "Shadow memory" bitmap. Shadow memories are used to keep track of some information about program memory during its execution. Each bit (or several bits/bytes) of shadow memory is mapped to one or more bytes in main memory. Thus, information about red zones is kept in the shadow memory and checked on every memory access. Purify was the first tool that implemented this technique [12]. It allocates a small red-zone at the beginning and end of each block returned by malloc. This way it can catch array bounds violations in dynamically allocated arrays. To catch same violations in statically allocated arrays, purify separates those objects by inserting red-zones in between.

   Purify maps every byte in heap, stack, data, and BSS section to 2 bits of shadow memory, which it uses to encode the state of given byte. Since it keeps track at byte-level granularity, it can detect off-by-one byte-level errors, but not bit-level errors.

   Purify uses 3 states to distinguish between "unallocated" (unwritable/unreadable), "allocated but uninitialized" (writable but not readable) and "allocated and initialized" (writeable/readable). Red-zones are assigned state "unallocated", so reads from outside object's bounds are also detected. Purify uses state "allocated but uninitialized" to detect usage of uninitialized variables, which is kind of a vulnerability that can be exploited. Upon every function call, Purify sets state of new stack frame to "allocated but uninitialized" in order to detect uninitialized local variables.

   The problem with this approach, however, is that for every memory access the tool has to do one more memory access for metadata. This is a

huge overhead since memory accesses are very slow. Light-weight Bounds Checking (LBC) Solves this problem by writing a random value in red-zones. [11]. On every memory read operation, LBC compares this random value with the read value. If it doesn't match, this address could not have been in the red zone, and thus extra read from metadata is avoided. If it matches, it has to do additional read from metadata to check if it matches because it's a red-zone or because it's just coincidence. Since the random value is 8-bit, there is only 1/256 chance of such case.

However, notable problems with this approach are that, first of all, it can only detect illegal memory access that targets red-zone, and cannot detect illegal memory accesses that target another valid object. Those tools also fail to detect intra-object overflows since red-zones are not inserted inside objects between subobjects.

2. **Guard Pages**: Similar kind of functionality can be achieved with help from hardware, by means of paging. Just like red-zones, guard pages can be inserted between memory objects to detect the overflow. However, in this case, we won't have to monitor every memory access, which would increase performance. When the program tries to access guard page, it will trigger a page fault, which can be connected to runs some hook in the tool. This technique, however, suffers from large memory overhead, since it needs to insert the whole page between every two objects and the page size is fixed (typically 4KB). Microsoft has implemented guard page functionality in Windows, but we couldn't find exactly from which version [1].

3. **Per-pointer Bounds Tracking**: This kinds of tools keep additional information for every pointer. Such pointers are called fat pointers. Fat pointers store, along with actual pointer value, upper and lower bounds (or base/size) of the object pointed by the pointer. New pointers in C are created by two ways: either by malloc() or by taking the address of a global or stack-allocated variable using the & operator. When malloc is called, lower bound is the same as the returned pointer and higher bound is lower bound plus the allocated size, which is known on runtime. When an address of the variable is taken by using & operator, lower bound is the address of the variable and higher bound is lower bound plus the size of the object, which is known at compile time.

   This information is propagated as pointers get copied. When arithmetic operations are applied to a fat pointer, bounds information doesn't change, only the pointer value changes. Since creating of out-of-bound pointer is allowed by C semantics, no check is done after arithmetic operations. Note that array indexing is the same as applying arithmetic operations to pointers. Then, on every dereference the tool checks if pointer still points to address within the range. This is a straightforward method that can guarantee spacial memory safety. SafeC [3], CCured [21] and SoftBound [20] use this technique to search for vulnerabilities.

Since not all pointers need to be monitored, CCured created a type system, separate pointer kinds for different pointer usage modes. Some of the pointers are safe and require only NULL checking before dereference. CCured calls them SAFE pointers and doesn't do any checks when dereferencing them, thus producing zero overhead. Some of the pointers are involved in arithmetic operations and need bounds checking when dereferencing. They are called SEQ pointers. Rest of the pointers are involved in type casts, that makes it impossible for CCured to know the type of the referred object at compile type. Those pointers are called WILD pointers and need bounds checking and type checking when dereferencing. CCured uses annotations to qualify pointer types, but they also have a pointer-kind inference algorithm that performs analysis to discover the qualifiers for pointers in a program. This way, they reduce overhead by adding checks to only suspicious pointers.

SoftBound, however, claims that the biggest disadvantage of CCured is that it changes memory layout in programmer-visible ways, thus making it hard to interface with library codes. SoftBound's approach is similar to CCured in that it's also a compile-time transformation by inserting bounds checks to enforce complete spatial memory safety. Application code after SoftBound's transformations, however, is highly compatible with existing code because its disjoint metadata allows arbitrary casts and avoids changes in memory layout. It uses a table data structure to map pointers to their metadata. Since bounds information should also travel with pointers when passed to functions, SoftBound transforms every function by adding base and bound arguments at the end of the function for each pointer argument. Functions returning pointer now return three-element structures. Apart from those, SoftBound can also detect intra-object overflows by adjusting the pointer bounds to become bounds of the subobject when the pointer is cast from object to subobject.

The disadvantage of Per-pointer Bounds Tracking is that it won't generally work with non-instrumented libraries because they will not propagate and update bounds information correctly. Another disadvantage is high overhead.

4. **Per-object Bounds Tracking**: Instead of storing metadata for every pointer, tools like Jones and Kelly bounds checker (J&K) [15] store metadata for each object instead. This way checked code can inter-operate with unchecked code without restriction, without interface problems, with some effective checking, and without false alarms. Those tools don't need to instrument pointer creations and assignments, but only malloc/free. They keep the table of metadata for each allocated object. With this table, the tool can map a pointer to the pointed object's metadata, that contains base, size, and some additional optional information. The tool has to instrument every arithmetic operation on a pointer because a pointer is not allowed to point to another object. Otherwise, the tool will fail to look up

for the correct object's metadata. However, the problem arises with codes like this:

```
int *i;
int *a = (int *) malloc (64 * sizeof(int));

for (i = a; i < &a[64]; i++)
{
  // Do something with *i
}
```

Listing 2: Example of harmless invalid pointer.

Problem with this code is that `a[64]` is invalid, and thus, `i` points to invalid memory before exiting the loop. The C standard only allows out-of-bounds pointers to one element past the end of an array, but many programs violate this and generate illegal but harmless out-of-bounds pointers that they never dereference.

To overcome the problem, J&K pads every object in memory by at least one byte. This, however, is not possible in all cases. For example, we cannot add padding in function parameters because if the function is unchecked then function call becomes incompatible. J&K resolves it by flagging function parameters and treating them specially. Note that in this case checking becomes incomplete: the function may alter pointer by arithmetic operation and tool won't be able to detect it. Another case where the tool will not be able to do the tracking is when the function returns a pointer to an object that was created in unchecked code.

J&K gives warning when the pointer becomes out of bounds and stops tracking it. Thus, it cannot detect when an invalid pointer becomes valid again. CRED is another per-object bounds tracking tool that overcomes this disadvanage [24]. CRED creates a special object (called OOB object) for every out-of-bounds pointer and points those pointers to corresponding OOB objects. OOB objects keep the actual value of the pointer along with information about the referred object. OOB objects get propagated just like any other data but when it gets used as an address, it gets replaced by actual out-of-bounds pointer address and gets bounds-checked before dereferencing.

Baggy Bounds Checking (BBC) [2] achieves several improvements over J&K and CRED. First, it optimizes the overhead created by the table that keeps the start and size of the objects (bounds table). BBC achieves it by padding and aligning objects to powers of two. This enables storing bounds information in only one byte, compared to previous techniques which need at least eight. This is done by storing only the logarithm of

object size in bounds table.

$$e = log_2(size);$$

This number can later be used to recover the object size and base pointer:

$$size = 1 << e;$$
$$base = p \ \& \quad \sim (size - 1);$$

Thus, BBC partitions memory in chunks of $slot\_size$ bytes. Bounds table has an entry for each slot of 1 byte, thus memory overhead is $1/slot\_size$. $slot\_size$ can be tuned to balance wasted memory in paddings and table size. Memory accesses are also fast since index can be derived by just shifting the address to the right by $log_2(slot\_size)$. Thus, bounds info is retrieved with only one memory access.

The second improvement of BBC over CRED is that it achieves support for out-of-bounds pointers without creating OOB objects for each pointer. Out-of-bounds pointers within $slot\_size/2$ bytes from the original object can be handled as follows: First, BBC sets MSB of the pointer to 1 when the pointer goes out of bound. By use of memory protection hardware and restricting whole program to lower half of the memory (where MSB is 0), BBC prevents OOB pointers to be dereferenced without instrumenting dereferences. Recovering original object from OOB pointer is also easy: since all objects are aligned to slot boundaries, BBC just checks if OOB pointer lies in top or bottom half of the slot. If OOB pointer lies in the top half of the slot, that would mean that referred slot is the one before, while if it lies in the lower half, the referred slot would be the one after.

1.3 Case studies

In this section, we will take a more detailed look at some of the commonly used memory error-checking tools.

1.3.1 AddressSanitizer

AddressSanitizer (ASan) is the most widely adopted sanitizer. It's an open-source programming tool originally created by Google [27] that detects memory corruption bugs by doing compile-time instrumentation. ASan is currently implemented in Clang (starting from version 3.1), GCC (starting from version 4.8) and Xcode (starting from version 7.0). Chromium and Firefox developers are active users of ASan. The average slowdown of the instrumented program is 2x. Asan can detect use-after-free, heap/stack/global overflows, use-after-returns, use-after-scopes, initialization order bugs and memory leaks.

ASan uses several techniques to hunt for different bugs. It uses shadow memory to keep track of metadata. Every 8 bytes of application memory get mapped to 1 byte in a shadow memory. ASan replaces *malloc* to place red-zones around the allocated region, which corresponds to writing special values in the corresponding place in shadow memory. It also replaces $free$ to replace whole freed region with red zone. This way ASan can also detect use-after-free bugs. In order to catch stack buffer overflows, ASan also places red zones on the stack. It instruments memory accesses by checking address validity before dereferencing.

1.3.2 Valgrind

Valgrind is a dynamic binary instrumentation (DBI) framework [22]. It uses dynamic binary re-compilation, which means, when it runs, it loads the client program and disassembles the code blocks into an intermediate representation (IR). All tools that are built on top of Valgrind's framework work on this IR level. When instrumentation/transformations are done, IR gets converted by the core back into machine code. Thus, only this re-compiled code is run. Valgrind runs client application on a simulated or guest CPU, which has virtual registers. Those registers may reside in one of the actual registers or may be spilled to memory.

Memcheck is a memory error detection tool that is built on top of Valgrind's framework. It can detect heap/stack overflows, use-after-frees, usage of uninitialized values, double frees, memory leaks and some other problems. Memcheck works by mapping one bit, called Valid-address bit (A) to every byte in memory and one bit, called Valid-value (V) bit to every bit in memory and registers. A bit indicates if a program can do read/write there and thus is checked on every read/write. Those bits get set in those cases:

1. On startup, A bits get set for global variables
2. When calling *malloc*, A bits get set for allocated memory
3. When moving $\$SP$. A bits are set only for the area between stack base and $\$SP$
4. Some other specific cases.

This makes it possible to detect invalid reads and writes. V bit is used to detect accesses to uninitialized but valid memory. Those bits get propagates when data in the program gets copied around. Checking V bits when copying won't work because when structures have some padding inside, they also get copied when the whole structure gets copied and padding bytes are usually uninitialized. Thus, they are only checked when it may affect an application's externally-visible behavior. Such cases are when a value is used as a memory address, when control flow decision depends on it and when it's used as an argument in some syscall. When the data gets copied, V bits get propagated without any checks.

Chapter 2

Background

We have implemented a tool KATU that detects unbounded memory accesses using dynamic taint analysis. To achieve it, we have used DynamoRIO, a dynamic binary instrumentation tool platform [4]. It supports code transformations while it executes and exports an interface for building plugins. Our tool uses dynamic taint analysis technique to track all the so-called tainted data that application reads from various locations (command-line arguments, files, sockets etc) and propagates. Apart from propagation, we process all the conditional branches that depend on those tainted values and whether or not the conditional branch was taken[1]. Some of those conditional checks will bound the possible value set of the tainted value for given control-path. Then we use this boundedness information when application tries to access memory that depends on tainted variable or call some function from libc that has tainted value as an argument. Our reasoning is that if the address, that we are reading from/writing to, depends on the tainted value that is not bound, we have found a vulnerability: a read/write primitive that can be exploited. In the same way, if we call some function from libc (e.g. *memcpy*) that has source or destination or even size argument dependent on unbounded tainted value, we have found a vulnerability. Our tool then issues a warning and provides as much info as it can, depending on whether debugging symbols are available and on whether or not the executable was stripped.

2.1 Dynamic Taint Analysis

The purpose of Dynamic Taint Analysis (DTA) is to track information flow between sources and sinks [25]. An application has several ways of reading inputs from the user (e.g reading from the file, from the socket, from command-line and so on). Some of them can be "trusted", which means that an external user has no influence on that data. This can be, for example, read-only configuration file.

[1] Except conditional branches, x86-64 also supports conditional moves. Our tool keeps track of those as well.

Other sources, which are called "untrusted", can be influenced by an external user by some means. We mark all the data that is read from those sources as "tainted". All other data is "untainted".

Taint propagation rules then dictate how exactly the taint is propagated. More specifically, it dictates in which cases, when some operation is applied to tainted or untainted values, the result should be tainted. Those rules depend on what exactly the tool using DTA is trying to achieve and vary from one tool to another. For example, every taint policy that we have seen marks result of *increment* operation tainted if the argument is tainted. This makes clear sense, since if the input is user-controlled and can become any value, that value +1 can also become any value[2]. On the other hand, some tools may taint result of the division of tainted number by 1024 while others may not, because the attacker loses 'part of it's control' on the value. Apart from those, there are some cases where the result of the operation is a constant value and we definitely shouldn't propagate taint in those cases. Sometimes it can be challenging to detect such a case. One of the examples of such case is *xor*. If we *xor* any number with itself the result is zero, and thus the result should not be tainted even if operand was tainted. Note that many compilers use this trick when they want to clear out the register. Many DTA tools (e.g. Dytan [6], libdft [16] etc) correctly handle this case but there are other, more challenging cases as well.

Taint checking rules dictate when taint information is actually used. For example, when program counter points to instructions that are tainted, we can deduce that software is executing injected code. Some of the DTA tools also check if tainted arguments are supplied to some sensitive syscall, like *exec*. Some other cases are checked depending on what exactly the tool using DTA is trying to achieve. For example, some tools may check whether the pointer that we are dereferencing is tainted or not.

Generally, two kinds of errors occur in DTA tools. One of them is when the tool marks something as tainted while it shouldn't. This is called *overtainting* and usually leads to a lot of variables becoming wrongly tainted, which eventually produces false positives. Another error is where the tool fails to detect the information flow and doesn't propagate the taint. This is called *undertainting* and the tool may not be able to detect some attacks if it happens. Thus, it usually leads to false negatives.

Another difference between the DTA tools is that while some of them (e.g. Dytan [6], libdft [16] etc) do application-level taint analysis, others (e.g. TEMU [29]) offer the whole-system dynamic taint analysis. Such tools have a whole-system view and are thus able to keep track of taint information as it gets propagated inside operating system kernel and between multiple processes. This is necessary to be able to detect attacks that involve multiple processes. Plus, it offers better isolation between the tool and the application and thus it's more difficult for the application to interfere with the tool. We will discuss how this interference can cause problems in DynamoRIO in the corresponding section.

[2]Assuming we don't have saturation arithmetic.

2.1.1 Propagation policies

Taint propagation policy determines under which conditions the taint gets propagated and under which it doesn't. In this subsection, we will discuss some of the cases that cause problems in DTA tools. As a first problem, let's ask ourselves a question: Shall we taint the value read from the memory if the pointer that was used was tainted? To illustrate it better, let's take a look at the following code:

```
// Assume we have defined char *A[] and tainted integer i.
char *args[2];
args[0] = A[i];
args[1] = NULL;

execvp(A[i], args);
```

Listing 3: First example of case-dependent vulnerability.

This code corresponds to reading from memory address via tainted pointer pointing to `A+i`. Do we have a vulnerability here? Unfortunately, the right answer is 'it depends'. If an attacker has full control over i, it has full control over `A+i`. Then, it can search in memory for example for "/bin/bash" and choose such a value for `i` so that `A+i` points to that string. On the other hand, `i` may have its bounds checked somewhere in the program, in which case given code is completely safe. It may also be that `A` has a size of 65535 and `i` is derived from some `short`. As you can see, to be able to say if the given code is vulnerable or not we need to have some additional information, like the size of A and possible value-set of `i`. To make this problem even harder let's take a look at the following code:

```
// Assume we have defined array char A[256] and tainted array char c[16].
int i;
char b[16];
for (i = 0; i < 16; i++)
   b[i] = A[c[i]];
```

Listing 4: Second example of case-dependent vulnerability.

Shall we taint b? Looking at the previous example we decided that if possible value-set of array index fits with array bounds it should be safe to not taint the result, but unfortunately, that's not always the case. Let's look at how *toupper* is implemented in libc:

```
return __c >= -128 && __c < 256 ? (*__ctype_toupper_loc ())[__c] : __c;
```

Listing 5: Implementation of *toupper*.

Which means that the example above can be just converting a string to its lowercase version, which still leaves notable influence over b to the attacker. For many programs, thus, it will be correct if we propagate taint in such case.

Another problem in DTA tools is the so-called "implicit flow of taint". Imagine we are given such code:

```
// Assume we have tainted int i;
int j;
if (i == 1)
    j = 1;
else if (i == 2)
    j = 2;
else if (i == 3)
    j = 3;
...
```

Listing 6: Example of control dependency.

What we have here is a control dependency, `j` is control-dependent on `i`. Pure DTA tools will not be able to detect implicit flows because they only follow the given path in a control-flow graph and to be able to detect control dependencies tool needs to reason about multiple possible paths. Thus, the tool will not propagate taint from `i` to `j`. As a result, the tool may undertaint. Such codes can be used to translate data from one format to another. For example, Windows keyboard driver propagates input in such fashion.

Another problem arises when we are dealing with cryptographic libraries. Imagine we have a function that encrypts its argument. Since modern encryption functions usually manipulate bits and bytes in an argument in very complicated ways, it's hard to keep track of taint when encrypting. But the problem is that, if we track taint correctly and find that returned, encrypted string is also tainted, that might not be quite what we want. Since the idea of encryption is to make it hard to go from an encrypted string back to the original, an attacker will have a very sophisticated control on an encrypted string by manipulating the argument. Thus, it might be better if we remove the taint in such cases.

2.1.2 Measuring influence

As we can see, having taintedness values in binary form is problematic sometimes. For example if `i` is tainted, we will propagate taint to `j = i & 0xF`, even though we have a very little control over `j`. Similar situation can happen in following code:

```
int j = 0;
if (i > 0 && i < 10)
{
  j += i;
}
```

Listing 7: Example of limited Influence.

In this case, `i` has very little influence on `j` but we still propagate the taint. To overcome the problem Newsome et al. [23] have proposed an approach to quantitatively measure the amount of control that inputs have over the tainted values. They call it an *influence* and base it on channel capacity, which is a measure of the maximum information that can be transmitted on a channel. Quantitatively, influence is defined as $\log_2$ of size of the set of all possible values (feasible value set). It's easy to see that, under such definition, in both examples `i` has a very limited influence on `j`.

This technique also helps with other above-mentioned problems. In case of table lookups, influence becomes $\log_2$ of the number of unique entries in the table. This information will help us decide if the taint should be propagated or not. For example, in case of *toupper* we will have mostly different values in the table and since ration of unique entries in that table over the size of all possible value set in *char* (255) will be big, we will propagate the taint. On the other hand, If we have large table of C-strings for example, the ration of unique entries over all possible strings of given length will be small and we will not propagate the taint[3].

The amount of influence that the tool should allow without issuing a warning has no perfect value. Through experiments, they found that the largest safe influence was 4.17 bits, while the smallest vulnerable influence was about 26 bits (out of 32).

2.2 DynamoRIO

In this section, we will discuss DynamoRIO, a Dynamic Instrumentation Tool Platform that we have used to implement our tool. It was implemented by

[3]This of course also depends on the length of the strings. If we have all possible strings of length 5, for example, then the ratio will be one and we will have to propagate the taint.

Derek Bruening in 2004 as part of his Ph.D. Thesis. It stemmed from Dynamo dynamic optimization system developed at Hewlett-Packard. Later it has become open source under BSD-style license and now can be freely accessed and contributed to via https://github.com/DynamoRIO.

DynamoRIO is a platform that takes an executable and executes it instruction by instruction. Before executing, it gives its plugins the possibility to observe and change any instruction. To understand how DynamoRIO works we first have to understand the concept of *basic block*. A basic block is a sequence of instructions that ends with an unconditional control-transfer instruction (CTI). For example, the following sequence of instructions is a basic block:

```
push   rbx
mov    r14, rsi
sub    rsp, 0x8,
sar    rbp, 0x3,
call   400390
```

Listing 8: Basic block.

DynamoRIO stores basic blocks in *code cache*. It is the location where an application is executed from. Figure 2.1 shows the control flow between the components of DynamoRIO. It executes the application in a given way: every time when some instruction from the application has to be executed it first checks if basic block containing that instruction is already in the code cache. If it is, DynamoRIO just jumps to that instruction and basic block gets executed. If it is not, DynamoRIO reads following instructions until it reaches CTI (to have a basic block). Then it modifies the CTI so that DynamoRIO can regain control when basic block finishes executing. After this modification, it places the block in the code cache and jumps to the first instruction. After the last instruction of the basic block gets executed, DynamoRIO saves application's machine state and gains control (this is called context switch)[4]. This mechanism of executing gives huge performance improvements over executing by emulation, instruction by instruction because in this case the code is still executed natively, but from the code cache.

2.2.1 Optimizations

As mentioned in the previous section, concept of a basic block gives significant performance improvement over instruction by instruction emulation. However, there is still quite a big room for improvement. For example, if one basic block ends with a direct jump to another, DynamoRIO doesn't have to do a context switch. Instead, it can replace the last instruction in the first basic

[4]Note that this is the simplified mechanism of how DynamoRIO works that is enough for our needs.

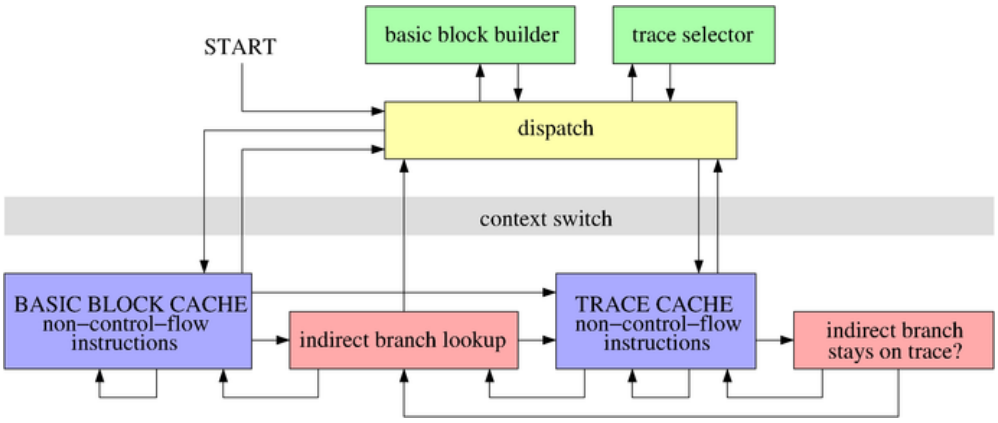


Figure 2.1: DynamoRIO architecture.

block to jump directly to the second one. This is also a significant performance improvement as an expensive operation is replaced by a single instruction. We cannot do the same for indirect branches though, since their destinations may vary. Original branch targets addresses are used when application stores branch targets. When jumping, DynamoRIO does a quick lookup of corresponding basic blocks in a special hashtable and jumps to it. Translation of indirect branches, however, is still the largest overhead in DynamoRIO.

Another improvement that DynamoRIO does is gluing together basic blocks that are executed together frequently. They are called "Traces". Elimination of inter-block branches gives significant performance improvements. DynamoRIO also 'remembers' most popular destination for indirect branches and checks it first before doing the hash lookup.

2.2.2 Transparency

When the program is being executed by DynamoRIO, the program should 'feel' like it is executing natively. This is challenging since DynamoRIO has to modify a lot of things to have execution under control. In order to achieve such transparency, DynamoRIO should make as small assumptions about stack/heap usage or OS interfaces as possible. It should not change parts of the application whenever possible, and when impossible it should make changes so the program doesn't notice them. Some of the biggest transparency-related issues and their solutions are:

1. **Resource usage conflict**: Ideally resources used by DynamoRIO and application should be disjoint but that's not always possible. For example, when an application has issued some call to a library function, then control is transferred to DynamoRIO and it executes the same call in the same library, re-entrancy issues will occur, which may lead to the wrong execution of the program. The solution is a private loader for libraries

which DynamoRIO uses, which loads a separate copy of the libraries. Not to interfere with data layout of the application, DynamoRIO uses separate heap allocation mechanisms. It obtains memory directly by system calls and uses internal memory manager to manage it.

2. **Using application stack**: Application stack must be kept the way it is natively. DynamoRIO cannot use application stack for scratch space because some applications like Microsoft Office uses data beyond the top of the stack. Thus, DynamoRIO uses a private stack for each thread.

3. **Threading**: DynamoRIO doesn't create any additional threads and it's executed by application's threads after a context switch. Threading library on Linux, LinuxThreads, uses stack to locate thread-local memory. It overrides weak symbols in C library by thread-aware routines. Using those routines by DynamoRIO won't work because it does not know about DynamoRIO's private stack. For future, the best solution would be to not use C library functions at all but currently, DynamoRIO overcomes this problem by using lower-level C library routines directly (e.g. *__libc_open/__libc_read*).

4. **Exception generation**: Since DynamoRIO must act like it's not there, an application bug that writes to invalid memory that would have generated an exception should still generate an exception even if DynamoRIO has placed some internal data at that location. However, it won't happen because for OS it's a memory that the process has allocated. To overcome this problem, DynamoRIO divides execution into two modes: DynamoRIO mode, which corresponds to DynamoRIO's code and application mode, which corresponds to code cache and DynamoRIO-generated routines that are executed in the code cache. DynamoRIO makes code pages of both modes write-protected, application data writeable from both modes and tries to protect DynamoRIO's data and code cache from getting overwritten in application mode by making it read-only. However, there are some complications because in some cases some of the DynamoRIO's data needs to be modified from DynamoRIO-generated routines that reside in the code cache.

5. **Address translation**: Even though the code is moved to the code cache, we still need to keep consistency between code cache and the original code. For example, every address that gets manipulated by code will remain original application address. As we have already mentioned, every indirect branch must be looked up to jump to the corresponding address in the code cache. Opposite situations may also occur, where DynamoRIO has to translate an address from the code cache to address in the original application. Such case will happen for example when OS gives machine context to signal or exception handler. In such case, address and registers in handed context will have originated from the code cache and DynamoRIO must translate it so that it looks like it happened in the original code.

6. **Error transparency**: Occurring errors should also be transparent. When e.g. illegal instruction error occurs, it shouldn't cause DynamoRIO to crash. Instead, DynamoRIO must be able to detect such error and deliver it to the application.

2.2.3 Implementation

Since DynamoRIO has to support quick insertions and deletions of instructions, it represents instructions as a linked list of *Instr* data structures. *Instr* can represent either single or several un-decoded instructions. Since x86-64 has complicated, variable-length instructions, DynamoRIO only decodes the ones that are needed for performance reasons. Thus, it offers several level-of-detail instruction representations, depending on the needs, where the lowest level (level 0) is raw instruction bytes of several instructions, only knowing bounds of the last instruction and the highest level (level 3) is fully decoded instructions with opcode, prefix, operand and affected EFLAGS information. Note that fully decoded ones can be modified (or new, made by DynamoRIO) and thus have invalid raw bits (level 4). Thus, they have to be encoded by DynamoRIO to become valid x86-64 instructions. When reading a new basic block, DynamoRIO only needs to decode the last instruction fully (the CTI) to do the modifications before placing it in the code cache. Thus, at that moment we have two Instr's, one at level 0, containing all but the last instructions and one at level 3, containing the CTI.

To perform transparent operations inside application code, DynamoRIO needs a special space, called *scratch space*. It's used mostly to spill registers: To save application register temporarily while DynamoRIO uses that register. When finished, DynamoRIO restores the original value of the register from the scratch space. Implementation of scratch space depends on the architecture: If absolute addressing is possible, it could be simple memory accesses to some absolute address. If the absolute addressing is not possible, we can spill one register and use it as the base address of the scratch space.

DynamoRIO provides a mechanism that allows the application to communicate back to DynamoRIO. This is achieved by invoking the annotation macro, which when compiled corresponds to *nop* and thus has no effect on native execution, but when running under DynamoRIO, it detects the annotation macro and it gets replaced by a call to a handler that was registered for the annotation.

Chapter 3

Implementation

This chapter discusses the implementation of KATU. Section 3.1 describes the idea of our project. Section 3.2 describes the architecture of the project, including modules that we divided the system in, their purposes and communication. In Section 3.3 we will talk about some of the notable technical details of the tool.

3.1 KATU

Before we begin, we would like to point out that the way we use the word *boundedness* is different from how it's used in the world of mathematics. First of all, it's different because everything in the world of computers is bound by technical limits. In our case, a byte is always bound from both sides, by either 0 and 255 if we look at it as an unsigned number or by -128 and 127 if we look at it as a signed number. We call the byte *bounded* if it was affected by one or several instructions that narrowed its possible value set so much that we believe it has become safe to use in places where having an unbounded variable is a vulnerability. We call such places "unsafe". We will discuss more about when we believe the variable is safe in section 3.1.1.

To understand our idea better, let's start with a simple example:

```
int var1;
scanf("%d", &var1);

int var2 = 0;

if (var1 < 30)
{
  var2 = A[var1];
}

return 0;
```

Listing 9: Vulnerable application C code.

This code has a vulnerability. For the given if condition to get satisfied, `var1` can have any value from INT_MIN to 30. This means we can assign to `var2` from a huge area of the memory, possibly reading memory that contains sensitive information. Now let's look at possible assembly of the given code:

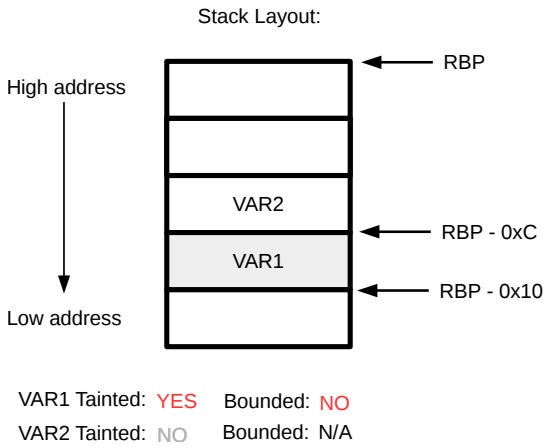
```
1:   lea    rax,[rbp-0x10]                  #40054e
2:   mov    rsi,rax                         #400552
3:   mov    edi,0x400614                    #400555
4:   mov    eax,0x0                         #40055a
5:   call   400430 <__isoc99_scanf@plt>     #40055f
6:   mov    DWORD PTR [rbp-0xc],0x0         #400564
7:   mov    eax,DWORD PTR [rbp-0x10]        #40056b
8:   cmp    eax,0x1d                        #40056e
9:   jg     400582                          #400571
10:  mov    eax,DWORD PTR [rbp-0x10]        #400573
11:  cdqe                                   #400576
12:  mov    eax,DWORD PTR [rax*4+0x601060]  #400578
13:  mov    DWORD PTR [rbp-0xc],eax         #40057f
14:  mov    eax,0x0                         #400582
15:  leave                                  #400587
16:  ret                                    #400588
```

Listing 10: Assembly of C code from Listing 9.

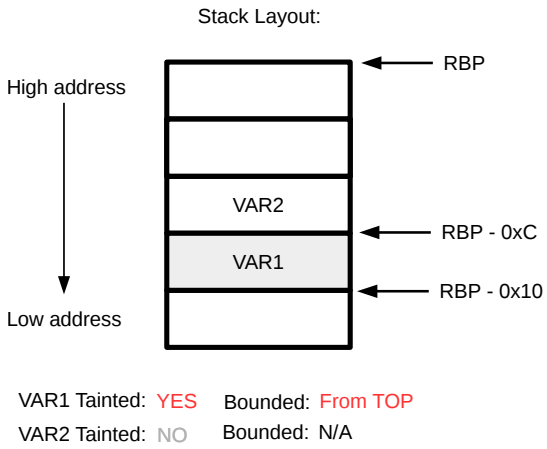
First four instructions correspond to setting up arguments to call to *scanf*. As you can see, it loads `var1`'s address to rax and then moves it to rsi. *Scanf* will then modify data at this address by writing there the read integer. Our

tool detects call to *scanf* and marks data at address specified by rsi as tainted.

By next instruction application stores 0 in rbp-0xc, which corresponds to `var2`. Thus, after the 6th line we have the following picture:



Then, it loads `var1` to *eax* and compares it to 0x1d. Let's say `var1` had value of 0x7, and since 0x7 is not greater then 0x1d the conditional jump wouldn't happen. At this point the tool realizes that, on one hand, for any value greater then 0x1d the next instructions would not get executed, and on the other hand, for any value less then 0x1d the next instructions would get executed before we reach another conditional statement. Thus the tool knows that, even though `var1` (and thus *eax*) has a value of 0x7, it could have any value below 0x1d and we would end up in exactly the same place. In other words, the tool knows that $var1 \in (-\infty, 0xd1)$. Thus, after executing the instruction on the 9th line, we have the following picture:



Note that even though the check was applied to *eax*, the tool knows that it was copied from rbp-0x10, so the same restrictions apply to both *eax* and rbp-0x10.

So the execution continues. After not taking the conditional branch we move `var1` to *eax* again, sign-extend it with *cdqe* and then we are accessing memory based on that. The tool knows that `var1` is not bound from the bottom, so it has detected a vulnerability and issues a warning. Note that when we executed the application with the tool, if had we entered some value larger then 0x1d the jump would have succeeded and the tool would have failed to detect the vulnerability because the vulnerable code wouldn't get executed anymore.

This example illustrates how our tool works. As the execution proceeds, we remember outcomes of all the conditional operations and shape the boundedness information accordingly.

We introduce the taint whenever we detect a *read* system call or call to *read*, *scanf* or *recv** functions in libc. We also taint command-line arguments passed to the application and the environment variables. To each read byte we associate a Unique ID (UID). For each UID we keep information about the origin (e.g. which file or socket was it read from) and the boundedness information. Since we want to keep track of the origins of taint as well, we keep track of fopen/open/accept functions and their returned values. We describe how we save information about the origins of UID in section 3.2.

We follow the application even when it executes functions from shared libraries. That is, we process instructions from shared libraries just like we do it from our application. The only exception is libc. Since libc is the library of standard functions and we have a summary of their effects, we can have a huge improvement in performance if we don't analyze the execution of those functions on instruction level. Instead, we can write separate function summary for all functions in libc. Such summary will summarize the effects of the function. For example, if we detect a call to *strcpy*, we can directly copy the taint from the source to the destination. Except for performance improvements, not all of those functions are implemented in a straightforward way. Some of them are quite tricky to analyze for the tool, and this feature also fixes that problem.

3.1.1 Propagation policy

Even though we associate UID to each read byte, we don't use it to mark tainted bytes. Instead, we use Taint ID (TID) to mark each tainted byte in memory and registers. We will best explain why we cannot associate UID to each byte in memory and registers and why we need TID via the following example:

```
char var1, var2;

scanf("%c %c", &var1, &var2);

char var3 = var1 + var2;
```

Listing 11: Example to illustrate the propagation policy.

Since we have read 2 bytes, we have 2 UID's, one for each variable. What do we do with `var3`? We need to somehow store information about `var1` and `var2` inside as well, since we will need it to calculate boundedness information. For example, if `var1` and `var2` get bounded from both sides at some point in the code, it would also mean that `var3` is bounded. To fix this problem, instead of associating UID with tainted bytes in memory and registers, we associate TID with them. New TID is created everytime new UID gets created or some operation happens between two tainted variables and to calculate the boundedness of the result we need the boundednesses of the arguments. TID has information about UID's that it was made from and operations that happened between those. For our example, 2 new TID's would get created after *scanf* returns (e.g. TID 1 and 2), both of them having single, their own UID's associated (TID 1 having UID 1 and TID 2 having UID 2) and one more TID would get created when addition happens, having both UID's associated with it as well as the plus operation (TID 3 with UID 1 + UID 2). We will explain how we use this information to calculate boundedness in section 3.2.7.

Note that we would not create a new TID if we had an addition of the tainted variable with constant or an untainted variable, because in those cases the boundedness still depends only on the tainted variable and thus the TID object would look exactly the same. So, in the given case, we just copy the TID from tainted variable to the destination.

In case of bigger variables, for example for integers, we associate separate TID's for each byte. When we read an integer with *scanf*, for example, 4 new UID's get introduced for each read byte, and 4 new TID's for each UID. When we do an operation on such variables, taint propagation happens on byte level. This makes sense because the boundedness information of any byte in the result of addition only depends on the boundednesses of corresponding bytes in the operand. We can safely ignore the carry bit information that gets propagated from less significant bytes to more significant ones because it is either +0 or +1 for the affected byte and +1 does not change the boundedness information. Note that this scheme is also correct for cases like when one operand has all bytes tainted but another one only several less significant bytes, possibly caused by extension from the smaller variable.

This scheme of byte-level propagation works on some instructions but not all. For addition, subtraction, *AND*, *OR* and *XOR* it works because any byte

gets affected by at most ± 1 carried from another byte, which does not change the boundedness information. This is, however, not the case for multiplication. Let's say we are multiplying two integers. We cannot say that the boundedness of the second byte of the result depends only on the boundednesses of second bytes of the operands simply because multiplication doesn't work that way. The second byte of the result is unbounded, for example, if the second byte of one of the operands is unbounded. It's also unbounded if first bytes of both operands are unbounded. For our needs, we made a small, harmless simplification and we assume that the result of the multiplication is bounded if all bytes of both operands are bounded.

Apart from introducing and propagating taint, we also sometimes need to change the boundedness information or remove it. We have already seen that conditional instructions are one type of instructions that change boundedness information. Some of the binary instructions also do so. For example, adding a tainted byte to non-tainted nor introduces new taint ID neither modifies it. It just copies TID from tainted operand byte to the result. Things are different, for example, if we AND tainted byte with non-tainted. In this case what happens to the result of AND instruction depends on the value of the untainted operand. If it's 0x0, result of the instruction is clearly untainted because it can only be 0x0. On the other hand, if its 0xFF, result of the instruction needs to have the same TID as the tainted one, because ANDing with 0xFF doesn't change the value. All the values between 0x0 and 0xFF need some thinking, experimenting and tweaking. Shall we still assume a byte to be unbounded if it was ANDed with some random value, for example, 0xC6? hard to say. Going back to our definition of boundedness, it's time to explain when and why we believe a tainted variable has become "safe".

To have some idea about when we can call a variable safe, we ran some experiments. We wanted to answer the following question:

> In real-world scenarios, what are the restricting instructions that get applied to tainted variables throughout their lives before they are used in 'unsafe' places, in which cases are those restricting instructions used and whether or not we should consider a variable to be bounded after those instructions.

Luckily, we got a simple and interesting picture. We saw that most of the instructions that would have been hard to analyze simply don't happen. There have been cases where GCC does an optimization which is very hard to analyze, but we'll talk about those cases and how to overcome them in section 3.3.8. Apart from that, we saw that, for example, a tainted variable never gets ANDed with 0xC6 before getting used in an unsafe place! Thus, we shouldn't care about such case. There are many other situations, which haven't happened through our examples, and in section 3.3.4 we will discuss how we handle such cases.

For AND operation where one of the operands is tainted and the other is not, we noticed that untainted operands have a simple and systematic pattern. In all the cases, their binary representation consists of several zeroes followed by

several ones. This is a simple trick GCC uses to ensure that a variable is in some range, if a range is a power of two. For example, for any unsigned `1`, `var1 & 0x7` clears out all the bits except last 3, which means that result of `var1 & 0x7` is in range 0 - 7. How many beginning zeroes are enough to assume the variable is bounded is a matter of tweaking. This is a very large and time-consuming part that goes beyond the scope of this Thesis. To get an optimal value, lots of commonly used libraries need to be analyzed for such cases. Since it would take us far away from our project, we chose those numbers based on a few examples that we have analyzed. This is a potential improvement that can be done in the future.

For OR operator we don't really care about the value of the untainted operand. It's true that if we OR tainted variable with untainted, we lose the influence and possible value set becomes narrower, just like in case of AND. However, the difference here is that we cannot use OR to limit tainted variable to some set of small numbers, just like we did with AND in the previous example. We can, theoretically, use OR to do the similar thing for negative numbers, but we could not detect GCC using OR operator in such fashion. Thus, we can ignore OR operation with untainted value. Similar kind of reasoning can be applied to XOR.

Things get a bit more complicated if we do AND, OR or XOR on two tainted variables. Before starting to think about what should happen in such case, we again wanted to see in what kind of situations such scenario would happen. As we could not see such scenario happening in real-world examples, we haven't implemented support for it now. Instead, we have postponed implementation to point when we detect it in some real-world application.

Another problematic case is shift operation. X86-64 supports several kinds of shifts (e.g. logical, arithmetic, circular) in both directions. As we have seen, shifts are mostly used in two situations: for bit manipulations and division/multiplications by the power of two. If shifting is being done by multiple of 8, we can just shift corresponding TID's of bytes as well. If it's not the power of 8, we have a situation which needs tweaking again. Exactly the same thing happens in case of division.

3.2 Architecture

KATU uses DynamoRIO as a binary instrumentation framework. We have tried to make it as decoupled and modular as possible, by dividing it into several modules. The architecture of our system can be seen in the Figure 3.1.

As you can see, our project consists of several modules, that are marked with dashed borders. In the middle we have DynamoRIO core, that we are using mostly for given two things:

1. When it encounters a new basic block it has to let *Analyzer* module do the instrumentation before placing it in a code cache.

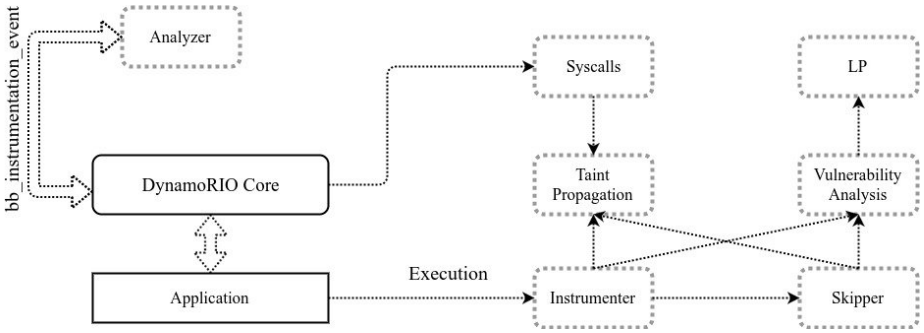


Figure 3.1: Architecture.

2. When detecting a system call, it has to tell *Syscalls* module about the arguments of the syscall and return value.

We will explain each of the modules in a greater in the following subsections.

3.2.1 Analyzer

Analyzer is connected to DynamoRIO's *drmgr_register_bb_instrumentation_event* hook, which is called before inserting a new basic block into the code cache. Thus, analyzer gets to see any application code before executing[1]. It iterates over all the instructions in the basic block, extracts opcode and calls the corresponding analyzing function. This is achieved via indexing opcode to an array of function pointers *instrFunctions* and thus is very fast.

Depending on the opcode we either ignore the instruction (e.g. *nop* or *inc*) or decode it, fetch operands and insert a corresponding *clean call* before that instruction. A clean call is a DynamoRIO's mechanism that inserts a transparent call to client routine. What it does behind the scenes is, it inserts a couple of instructions before the given instruction. Those new instructions save machine state, set up arguments to call the inserted routine, switch to a new, clean stack do the call. Note that we have to switch to a new stack for transparency reasons, since DynamoRIO doesn't make any assumptions about the validity of the stack in an application.

Inserted calls are very small, instruction-specific functions that reside in *Instrumenter* module. Let's take an *add* instruction from x86-64 as an example [14]. the called function needs to know where are the source operands and destination located. In Figure 3.2 we can see that *add* instruction has several different variants.

Gladly, DynamoRIO abstracts some of the information from actual opcodes

[1]Note that if the same code gets executed several times (e.g. loop or function), Analyzer will only get notified once.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8*.
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64*.
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16*.
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32*.
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64*.
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16*.
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32*.
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64*.
00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /r	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32*.
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64*.
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /r	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.
03 /r	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 /r	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Figure 3.2: Variations of *add* instruction [14].

by providing decoded sources and destination operands. As you can see, source operands can be either immediate, registry or memory location and the destination can be either registry or memory location. Apart from that, x86-64 has several different memory addressing modes, but they can all be united in two following forms:

1. **seg_reg:[base_reg + scale*index_reg + disp]** - Scaled indexed addressing mode. This mode does a memory read from the address that is calculated by the given formula. *seg_reg* here corresponds to the base address of the chosen segment. Base_reg and index_reg are values read from given registers. Those two registers together with scale correspond to so-called SIB byte. In this byte, 3 bits are used to store base_reg, 3 bits for index_reg and thus only 2 bits are left for scale. Thus, scale can only have 4 values: 1, 2, 4 or 8. Disp corresponds to displacement value and is either 0, 1, or 4 bytes long in 32/64bit addressing modes. When using this addressing mode, we have to be able to calculate the exact address from within the function. Thus, we have to pass segment register, base register, index register, scale and displacements as arguments to the routine. Note that we cannot fetch register values and calculate the address in Analyzer module when creating the basic blocks because we cannot know what the

values of the registers will be when we reach that instruction. Thus, we can only pass register name (or ID) to the routine, not the value that was there when the basic block was being made.

2. **seg_reg:[RIP + disp]** - RIP-relative addressing mode. This is the new addressing mode that is only available in 64-bit mode and makes it way easier to generate PIC code. To calculate the address, constant offset gets added to the address of current instruction (RIP). Seg_reg here has exactly the same purpose and disp corresponds to a signed 4-byte displacement constant. In this addressing mode, however, we can do the decoding in analyzer module because we know what *RIP* will be when the code gets executed and we also know that it won't change. Thus in given case we decode the address and tell the routine already calculated memory address.

Seg_reg defaults to DS (Data Segment) if the calculation is based on one of EAX, EBX, ECX, EDX, ESI, EDI registers and to SS (Stack Segment) if it's based on EBP or ESP. However, segmentation became marginal in x64. Linux, for example, uses segmentation in a very limited way. On Linux, code, data and stack segments all have the base address set to 0x0 and cover the whole address range. We will discuss this in more detail in later sections.

Another form of addressing can be obtained by taking the subset of the scaled indexed addressing mode. E.g, we can skip displacement or scale*index_reg.

Any code gets into the *Analyzer* module only once but can get executed many times. Thus, for performance reasons, if some calculation can be done in both *Analyzer* and *Instrumenter* modules, we'd better do it in *Analyzer*. As an example, let's take Load Effective Address (*lea*) instruction. This instruction computes the effective address of the source argument, which is a memory address. As we have seen, memory addresses come in two different forms (addressing modes), let's take the scaled indexed addressing mode for example. Thus, we have such an instruction:

```
lea dst_reg, seg_reg:[base_reg+index_reg*scale+disp]
```

This basically corresponds to $dst\_reg = seg\_base + base\_reg + index\_reg * scale + disp$. As we have already mentioned, other addressing modes correspond to subsets of this. That is, we may have for example $dst\_reg = seg\_base + disp$, or $dst\_reg = index\_reg * scale$. On one hand, we can have a general routine for all possible versions of *lea* in the *Instrumenter* module, decode there and take the corresponding action, or we can do the decoding in *Analyzer* and add the call to the corresponding routine. We chose the latter because, first of all, we have to do decoding only once per instruction (even if it gets executed many times), and second of all, we will have simple and reusable routines in *Instrumenter* module. For example, in case of $dst\_reg = index\_reg * scale$ we only have to propagate taint from $index\_reg$ to dst_reg. Thus, we can use the same routine for cases like $mov\ dst\_reg, src\_reg$.

3.2.2 Instrumenter

Instrumenter module consists of many small routines, that get called from within the application via *clean call* mechanisms inserted by *Analyzer* module. These routines can be split into several groups. Some of them only do the taint propagation or modification (e.g. *restrictors* and *shifters*). For those instructions, *Instrumenter* asks *Taint Propagation* module to do it.

Another group is jump instructions. For jumps, we need to do several types of operations. First of all, since jump is an "unsafe instruction", we need to check that, if the destination address is tainted, it's bounded. If the destination address of an indirect jump is unbounded, then this is a control-flow hijacking vulnerability. *Vulnerability analysis* module does this check for us. Another thing that we do with jumps is, we check the destination address of the jump and ask the *Skipper* module if this jump corresponds to call to one of the functions that we must skip processing. If so, we switch to "skipping" mode, call the "entrance" function from the *Skipper* module and save the pointer to "exit" function from the *Skipper* module as well as the address of the instruction that must get executed when this call finishes. We will talk about "entrance" and "exit" functions in section 3.2.5. If we are already in the skipping mode, we check if the jump corresponds to the return from the skipped function (by comparing the destination of jump and the address of instruction that we had saved before entering it), and if it does, we call the saved "exit" function and switch back to normal mode.

Another group of instructions consists of *comparators* and corresponding conditional instructions. When we detect a conditional operation, we need to know which instruction the conditional operation depends on. Thus, we need to know which instruction was the last one to update EFLAGS. To be able to do it, we always keep track of which was the last instruction that modified EFLAGS and what did the instruction operate on (TID's). Some instructions are made specially for comparison and don't do anything else except updating EFLAGS. Such instructions are *cmp* and *test* on x86-64. Other instructions, like arithmetic operations, also update EFLAGS. Later, when we detect a conditional instruction, we update the bounds informaton accordingly.

For example, imagine EAX is tainted and we have a given code:

```
cmp    eax,0x7
jle    400635 <main+0x3f>
```

After the first instruction, *cmp*, *Instrumenter* will know that the recent comparison was between tainted register EAX and some untainted value. When conditional jump gets executed, it will first check if it would succeed or not. Note that to know whether or not the conditional instruction will happen we need to know the value of EFLAGS register and we don't keep track of EFLAGS explicitly. Rather, when the clean call happens, EFLAGS register is saved as part of the context and it can be retrieved from within the call. Let's assume

the conditional jump succeeds. It means that this conditional instruction has bounded the TID of EAX from above. Note that we could have also had a comparison of two tainted registers. The way we keep track of those bounds and then use them will be discussed in *Vulnerability Analysis* section.

The conditional operation doesn't necessarily depend on comparison instruction. For example, the given code:

```
if (--a >= 0)
```

can get compiled into the following assembly:

```
sub     DWORD PTR [rsp+0xc],0x1
js      40044d <main+0x1d>
```

Here the conditional jump depends on the result of *sub* instruction. In order to be able to correctly detect imposed restrictions for such cases, we keep track of all the instructions that update EFLAGS.

Another instruction that requires special attention is multiplication. As we have discussed in the *Theory* section, we assume that the result of the multiplication is bounded if all the bytes in both operands are bounded. We have implemented this by marking all bytes of the result with new TID that is calculated as if all bytes of both operands were added. This works because the sum of the independent numbers is bounded if and only if all of the numbers are bounded. Besides, we also have to be careful when we are multiplying with a negative number, because if the number was bounded from above but unbounded from below, after multiplying by the negative number it becomes unbounded from above but bounded from below. Thus, if we multiply two tainted numbers and both of them are tainted from above, we cannot say that the result will also be tainted from above since we don't know if one of them can get negative. This case needs a special handling which will be implemented in upcoming versions.

3.2.3 Taint Propagation

Taint Propagation module is responsible for all sorts of things related to TID propagation. As we have seen, *Instrumenter* module asks *Taint Propagation* module to handle some kinds of instructions. We'll go through them here, step by step. Before propagating, since operands can be memory locations as well, we first have to decode the exact memory address. As we have seen in *Analyzer* section, an address can be, for example, in scaled indexed addressing mode. In *Taint Propagation* module we read the values of registers that got saved as part of the context when doing clean call and calculate the exact address. Note that this is also a potentially unsafe place because base_reg and index_reg can be

unbounded and this instruction can access any memory. Thus, before decoding, we ask *Vulnerability Analysis* module to check boundedness of given registers.

Simplest forms of instructions are *mov* instructions. They just take data from one location and copy to another. This is simple to handle since all we have to do is copy the corresponding TID's. Some *mov* instructions don't copy data from one location to another. Rather, they have the source operand hardcoded in them, which is also called an immediate value. Since immediate value is hardcoded and cannot be tainted, in such cases we just remove the taint from the destination location.

mov instructions come in two additional forms: zero extension and sign extension. They are used when converting small unsigned/signed numbers to bigger ones, correspondingly. For example, they will be used when we are converting a 1-byte integer to 4-byte one. In case of zero extension, the remaining 3 bytes will be filled with zeros, and thus, they should clearly be untainted. But, in the case of sign extension, they will all be filled with either all zeros or all ones. Still, however, we should mark them as untainted because they only have two possible values. Plus, those bytes will never get used separately (Theoretically, it's possible to manually write assembly code that will use those bytes separately but we are concentrating on practical cases in this Thesis).

This module is also responsible for processing restrictor and arithmetic instructions as we have discussed in section 3.1.1.

3.2.4 Syscalls

DynamoRIO's *drmgr* extension offers simplistic handling of system calls by attaching special hooks to pre-syscall and post-syscall. Since our thesis is related to applications written in C programming language and it offers wrappers to all the syscalls, it's very rare to still see explicit *syscall* assembly instruction in them. Still, we support explicit syscalls. For example, for *read* syscall, in pre-hook we fetch and store the arguments of the syscall, which are in this case the file descriptor and the address. Then, when syscall finishes executing, post-hook gets called which checks the returned value. This value for *read* syscall corresponds to the number of bytes read. Then we mark the read bytes with new taint values.

3.2.5 Skipper

As we have already discussed, it's a good idea to skip processing known and commonly used functions instruction by instruction. Rather, since we know what the function does, we can manually make appropriate changes to shadow memory. This is beneficial, on one hand, because the implementation of those functions can be quite complicated and by skipping their instruction-by-instruction analysis we are avoiding potential wrong tainting inside those functions. Apart from that, we get a huge performance improvement by doing so.

We achieve this functionality by attaching a hook to a module loader event. Then, when libc gets loaded, we search in it for the functions we want to skip. Once found, we remember it's address and associate with them pre and post functions, just like in case of system calls. Internally, we use a hashmap to effectively search for addresses later, when processing jump instructions.

For some functions like *fflush()*, *listen()* and so on, we associate empty calls, since those functions don't change anything with respect to taint.

For functions that copy the data, we copy the corresponding TID's. Apart from that, we check if the size argument is bounded. For ones that do the comparison, we check if the compared data turns out to be equal. If it doesn't, we don't do anything since it doesn't mean anything useful. On the other hand, if the data turns out to be equal to each other, we have to update TID's to be equal. If one of the operands was untainted, we also have to mark TID's of another operand to be bounded.

For other functions that open or read from a file, terminal or a socket, we keep track of the descriptors and handles and do the appropriate tainting. The simplest way to store and then retrieve origins for each UID would be to store either file descriptor (FD) or FILE* for each UID, and in some other location store origins for each FD/FILE*, but it will not work because an application may close FD and Linux may re-use the same file descriptor for a newly opened file. Thus, not all UID's that are linked to the same FD will be originated from the same file. To overcome this problem, we associate special ID for to each file that we open and connect this ID with UID. We associate information about the origin with this ID. Thus, for each UID we can easily retrieve information about the origin.

Then, when we read from descriptor or the handle, we create new UID and corresponding TID for each read byte and associate TID's with the bytes.

scanf requires a special attention. Our module needs to be able to know how many bytes get written in each argument provided. In order to do so, we have to parse the format string just like libc does and detect what are the types of the variables. For example, when we detect *%d*, we know that 4 bytes should be tainted at address of the corresponding argument.

Another commonly used library is libm, which has implementations of mathematical functions (trigonometric functions, hyperbolic functions, etc). We have decided to completely ignore calls to this library because on one hand they are very complicated to process instruction by instruction, and on the other hand, we assumed that it's very unlikely that the result of any of the functions that libm exports gets used in a memory access.

3.2.6 Vulnerability Analysis

This is probably the most important module, as it's responsible for detecting the vulnerabilities. Vulnerability, in our thesis, corresponds to an unbounded memory access or function call with an unbounded argument for some set of functions. Before we begin, we would like to point out how we store the boundedness information.

Every UID has a bitfield for boundedness information. First 3 bits there denote bounded from below, bounded from above and constant (constant means that it compared to be equal to some untainted data). In the simplest case, we read the data from somewhere and introduce corresponding UID/TID's. Then, when we want to bound one or several bytes, we check corresponding TID's, find that those TID's are associated with only one UID each and mark those UID's boundedness bitfields.

However, things get complicated if TID is associated with several UID's. For example, let's say it's a sum of several UID's. For this kind of bounds, we create a special object, called *Group restriction* (GR). GR basically consists of TID and bound type. We have implemented them as single linked lists. Thus, each GR object also contains a pointer to the next one. We associate a GR with every UID that it is related to. Thus, every UID has its own linked list of GRs, and each GR appears in linked lists of all the UIDs that it is related to. Since some GRs can appear in several linked lists and thus will have different pointers to the next element, we create separate copies of GRs for each linked list. In the *ILP* section, we will discuss how we use those GR's to calculate the boundedness information.

Thus, we have a mechanism to store boundedness information for a given TID. However, not all comparisons lead directly to such format. In some cases, we may have comparison of two tainted variables and let's say `var1 < var2`. In this case, we can convert given inequality to `var1 - var2 < 0`, create a new TID for `var1 - var2` and create a GR for it. In this way, we convert all sorts of such constraints to canonical form for us.

Now that we have bounds information in UID's and their GR's, we can detect whether the data is bounded. When TID is being used in an unsafe place, we first check if this TID is associated with single UID or several. If it's associated with single, we check UID's boundedness bitfield. If this UID is bound from both sides (or constant), we can say that this usage is safe and proceed. Otherwise, we cannot directly say that this access is unsafe. To illustrate this, let's look at the following example:

```
short var1, var2;
scanf("%d %d", &var1, &var2);

if (var1 < 10)
{
  if (var2 < 8 && var1 + var2 > 10)
  {
    int var3 = A[var1];
  }
}
```

Listing 12: Example when only UID's own boundedness is not enough to detect the bounds.

Let's assume that the program reads "6 6" and `var1` and `var2` get UID's and TID's 1, 2 and 3, 4 correspondingly. When the program reaches first if statement, the tool detects that `var1` is bounded from above. It checks `var1`'s TIDs, finds that they are associated with single UIDs and marks corresponding bits in 1 and 2 UIDs boundedness bitfields.

Exactly the same happens with `var2` in the first part of the second if statement. Then, when we come to the second part, we have to create a new TID, that will be the sum of `var1`'s and `var2`'s UIDs and associate boundedness information with it. As we have seen, when TID is associated with more than one UID, we have to create a GR.

After passing if statements, we have a memory access indexed by `var1`. TIDs of `var1` have single UIDs associated and those UIDs are only bounded from above. It may look like we have a vulnerability, but actually we don't. If we look at all the constraints that we have, including all the boundednesses of UIDs and GRs, we will notice that `var1` is actually bounded from both sides. If we try to make `var1` too small, we'll also have to make `var2` too big so that the sum of them is more then 10, but we can't make `var2` too big because it's also bounded from the above. Thus, `var1` is bounded from below as well. We will introduce a general solution to solve such cases in *LP* section.

A similar situation can happen if we are trying to access memory with TID that is associated with several UID's. Let's take a look at the following example:

```
short var1, var2;
scanf("%d %d", &var1, &var2);

if (var1 < 10 && var2 < 10)
{
  if (var1 > 0 && var2 > 0)
  {
    int var3 = A[var1 + var2];
  }
}
```

Listing 13: Example of memory access by sum of variables.

In this example we have memory indexed by TID of `var1 + var2`. In order to decide whether or not the access is safe, we have to check boundednesses of all the UID's associated with the TID's as well as all the related GR's. In the next section, we will show a systematic way to handle such situations.

Following the described methods, we have found some false positives in real-world applications. The problem is, some real-world applications (like *tcpdump*) use tainted byte as array index directly, without checking its bounds. Of course you don't need to check bounds of the variable if necessary bounds are already enforced by technical limits. That is, you don't need to check if a 1-byte variable is less then 256, because it always is. Thus, if you want to index 256-element array with 1-byte variable, you don't need to check for variable's bounds. This is exactly what has been causing false positives in our tests. To fix this, we have added following check before claiming that the vulnerability was found: We check how many bytes inside participating argument are unbounded and we only claim that we have found a vulnerability if the number of unbounded bytes is more then one. One may argue that the same goes for arrays having 65536 elements and getting indexed by two bytes. This is true, and if such thing happens the tool will produce the false positive. However, we haven't found such situation in any real-world application that we have tested.

3.2.7 Boundedness checking using LP

As we have seen in the previous chapter, we need a systematic method to solve the boundedness problem for an arbitrary number of related constraints. It's possible to have UID that itself is not bounded but is part of some other GR's that limit its range. To state the problem in a more general way, we have several constraints, which have form of inequalities and equalities, and a polynomial, and our aim is to decide whether or not this polynomial is bounded. Thus, we have a system of inequalities and equalities:

$$\begin{cases} uid_1 + uid_2 + uid_3 \leq 0 \\ uid_1 + uid_5 \geq 0 \\ uid_2 - uid_4 = 0 \end{cases}$$

And then we have something that we need to check boundedness of, for example:

$$uid_1 + uid_2$$

Note that uids don't have any coefficients in front. That's because we ignore multiplications by constant values because it doesn't change boundedness. The only thing we care about when multiplying by constant is the sign of the constant, and if it's negative we just change the sign of our uid.

Another important observation is that on the right side we have only zeroes, even though according the code the constraint was added for some other number most likely. The trick here is, since we only care about boundednesses and not about actual numbers, we can put zeroes there. We do not show the mathematical proof of why is it OK to do so here, but it's easy to see from the logical perspective that if the original equation could not bound the objective function, we can scale everything down by any amount and the objective function will still be unbound. Note that we only have $\leq$ and $\geq$, operators and not $<$ and $>$ ones.

This kind of problems are solved with methods of Linear Programming (LP). LP is a technique for solving such kind of functions:

$$\begin{gathered} minimize\ C^T X \\ subject\ to\ Ax < b \\ and\ x \geq 0 \end{gathered}$$

Note that the last one is not true for our case, since x can have any value, not only positive. To fix this, we can introduce new variables and substitute any variable x_i with y_{2*i} - y_{2*i+1}. Then, we can search for solution for $Y \geq 0$ without losing any precision, because any value of x_i can be represented by subtraction of two positive numbers. Thus, after this replacement, our example will become:

$$\begin{cases} y_2 - y_3 + y_4 - y_5 + y_6 - y_7 \leq 0 \\ y_2 - y_3 + y_{10} - y_{11} \geq 0 \\ y_4 - y_5 - y_8 + y_9 = 0 \end{cases}$$

$$Objective function : y_2 - y_3 + y_4 - y_5$$

The matrix C and A, in our case, have a very simple form: they are made of only 1's and -1's. Since we need to check boundednesses on both sides, we

have to solve two problems for each check: one for minimization and one for maximization.

So, we have managed to transform our problem to LP problem. The next problem is, how do we pick up the necessary constraints? Of course we can give the LP solver all the constraints that we have and ask it to optimize our objective function. It will work, but, of course it will be tremendous unnecessary work. We need to find all the constraints that are related to our objective function either directly or indirectly. By indirectly we mean that constraint may have none of its variables involved in the objective function, but they may be shaped by other constraints.

We find the related constraints as follows: At the beginning, we take the TID that we are trying to optimize and add its UID to the objective function. Then, we iterate through the list of associated TIDs and recursively call the same function. Thus, in the end, we have the objective function as a sum of involved UIDs. Then we copy all UIDs to a "related UIDs vector".

After this step, we recursively remove UID from the "related UIDs vector", check it's boundedness bitvector and if it has any bound, add the corresponding constraint. Then we iterate over all GRs of the UID and add all of them to the constraint list. Before adding, we 'decompose' GR to underlying UIDs just like we did when building objective function. While decomposing we also keep track of all the UIDs that we encounter. For each of them, we check if it has ever been in "related UIDs vector", and if it hasn't, we add. In this way, we first create a list of constraints related to UIDs in the objective function. Then, we create list of constraints related to existing constraints and so on.

LP solver is a heavyweight solution that puts a heavy penalty on the performance of our tool. We only use LP solver when there is a need. Before using it, we manually check if UIDs bitvector has all the necessary information to decide about the boundedness. If it does, we don't have to use LP solver anymore, since we know that the access is safe. Luckily, this is the case in the vast majority of the vulnerability checks of KATU.

To solve LP problems in KATU, we have used lp_solve, a free linear programming solver based on the revised simplex method. We discuss our plans regarding LP solver in chapter 6.

3.3 Technical Details

In this chapter, we will talk about various technical problems and tricks that we have used to make KATU work.

3.3.1 Shadow Memory

Shadow memory is an integral part of every DTA tool. As we have discussed, shadow memories are used to store some additional information about every memory location that the application uses. We only need to store one number per byte: TID. We have implemented shadow memory by simple hashtable. The

key is calculated by *address mod hashmap_size*. Since we use a hashtable, we need one more variable per each entry in the hashtable to detect hits and misses. For each key, we have a fixed amount of available slots to handle collisions. If collisions for some key happen more than some fixed number of times, the tool will report and stop working. Then we'll have to change the defined value for the maximum amount of slots and rebuild the tool, but because of the nature of the programs, this is very rare.

Limitation about the maximum number of slots exists because we have actually used two 2-dimensional arrays to implement the shadow memory. The first index corresponds to the key and the second one corresponds to 'attempt' in case of the collision. First array, address[TAINTMAP_NUM][TAINTMAP_SIZE] stores actual TIDs while second array, value[TAINTMAP_NUM][TAINTMAP_SIZE] is used to detect hits and misses by storing actual addresses. Thus, to get TID for address x, we do the following:

```
int index = 0;

while(value[index][x % TAINTMAP_SIZE] != -1 &&
          address[index][x % TAINTMAP_SIZE] != x &&
              index < TAINTMAP_NUM)
{
  index++;
}

if (index == TAINTMAP_NUM)
{
  DIE("ERROR! Shadow memory failure at [A]\n");
}

int tid = value[index][x % TAINTMAP_SIZE];
```

Listing 14: Calculating TID for given memory address.

As you can see, while loop iterates through collisions and ends when:

1. It detects that there is no such entry in the hashmap (*value[index][x % TAINTMAP_SIZE] == -1*)
2. It finds the correct entry (address[index][x % TAINTMAP_SIZE] == x)
3. Hashmap becomes full (index == TAINTMAP_NUM)

We find that this implementation is effective because calculating the key is very fast (modulo operator) and we have a very low rate of collisions.

Apart from the shadow memory, we need a shadow registry file, which will do exactly the same thing for registries. For this, we have use a simple 2-dimensional array again, but this time first index corresponds to the registry number and the second one corresponds to the byte inside the registry.

3.3.2 Segmentation issues

As we have already discussed, x86-64 supports memory segmentation, but in a limited way. In the beginning, segmentation was added to Intel 8086 to make it possible to access more than 64KB of the memory. By that time, segments were only identified with base addresses, they didn't have defined lengths and thus this version of segmentation didn't offer any protection. Later, in Intel 80286, Intel introduced a new, "protected mode" which added support for virtual memory and memory protection. x86-64 has largely dropped support for segmentation by forcing CS, SS, DS, and ES segments bases to 0 and limits to 2^{64}.

On 32 bit systems, Linux uses a so-called flat memory model, which means that segmentation is effectively disabled by setting base addresses of segments to 0 and limit to the end of memory. On x86-64, as we have seen, hardware does it. This means that, as segments have the base address of 0, we can ignore the seg_reg when calculating the address.

There is, however, one problem remaining. Both 32 and 64-bit systems allow FS and GS to have non-zero base addresses. They are used, for example, by GCC to implement thread-local storages. This means that we can no longer ignore seg_reg part when doing the address calculation. The problem is, to read the base address of the segment we need to switch to kernel mode because instructions that manipulate segment descriptors are privileged instructions. Since DS has the base address of 0 and limit of 2^{64}, any memory can be accessed through DS, including the ones that belong to the GS segment. Thus, we cannot use separate shadow memory for the GS segment.

Solving this problem was quite tricky. After some searching in IA-32 ABI, we have found that ABI enforces *%gs:0* to point to the base address of *%gs* segment [10]. For example, this code:

```
__thread int x = 0;

int *f1() {
    return &x;
}

int f2() {
   return x;
}
```

would get compiled into this:

```
<f1>:
  mov    rax,QWORD PTR fs:0x0
  add    rax,0xfffffffffffffffc
  ret

<f2>:
  mov    eax,DWORD PTR fs:0xfffffffffffffffc
  ret
```

As you can see from `f2`, variable `x` gets allocated at -4 offset in fs segment. In `f1` we want to get the address of variable `x` and as you can see it works by subtracting 4 from whatever's written in *fs:0*. We have used this trick as well in KATU: When we are calculating the address in segment fs, we read from *fs:0* and add it to the calculated address.

3.3.3 Problems in Skipper

Skipper module is responsible for skipping instruction-level processing of the well-known functions from libc and manually making the appropriate changes to the shadow memory. We have faced problems implementing tainting functions for the *realloc* function in libc. *Realloc* function is used to change the size of the allocated memory block. If the new size is bigger, libc will first check if it can just extend the memory block. If there are other things after the block, it will have to do a new allocation for requested size and copy the contents of the old memory there. We have to know which is the case because if *realloc* does a new allocation and copies old memory there, we also have to copy the taint information of the old memory to a new location. This, however, causes the problems because the tool doesn't know the size of the previous allocation.

Implementations of *malloc* that we saw usually allocate memory as the combination of the header (metadata like allocation size) and the actual space for the data. In such case, since a header has a constant size, it's easy to get the size of the allocation by just subtracting some constant number from the pointer that points to the beginning of the space for data (this is what *malloc* returns). However, since this part is implementation-dependant we didn't want to use this information. Instead, we have created a special hashtable that maps allocation pointers to their own sizes. We do appropriate changes to this hashtable on every call to *malloc/calloc/realloc/free*.

Note that there are some functions in libc that call malloc internally. One of them is *strdup*. We update the hashtable in case of those functions as well.

3.3.4 Lazy Implementation

There are lots of instructions on x86-64. Plus, as we saw in *Analyzer* module, each instruction has many different forms. Apart from that, all those instructions can be encountered in various different scenarios for us. For example, we

can have shift logical left (*shl*) instruction shifting register CL times and CL might be tainted, or we can have *bswap* instruction on an untainted register. Having a list of all x86-64 instructions and writing their taint propagation functions, in alphabetical order, for all scenarios didn't seem like a good idea to us. Instead, we took what we call a "lazy implementation" approach. The name comes from the family of "lazy" computer programming optimizations, where some sort of work is delayed to the very last moment until it's actually needed. Such optimizations are, for example, lazy evaluation and lazy binding.

In lazy Implementation, we don't implement support of any scenario for any instruction until we actually encounter the scenario. Thus, in the beginning, we had a lot of unimplemented cases. In all those cases we had a special define made for it, called FAIL():

```
#define FAIL() { dr_printf("FAIL! at %s:%d.\n", __FILE__, __LINE__); \
                 dump(); \
                 exit(-1); }
```

FAIL() writes to the terminal at which line it had happened, dumps some useful taint information (since we re-use FAIL() in some other cases as well) and exits the program. When some unimplemented case happened, the tool would stop and tell us at which line it had stopped. Then we would just implement the support for that case and rebuild the tool. Even though we have experimented with lots of real-world applications with our tool, there are still a lot of unimplemented cases, which means they either just don't happen in real-world applications or are very rare and happen only in some specific cases.

3.3.5 Logging

To help with debugging, we have implemented four different levels of logging: WARNING, NORMAL, DEBUG and DUMP. Each of them can be turned on and off easily for each module in the tool. By default, only WARNING logs are on. We only turn on the rest when we want to debug the module.

We have redirected all logs to NSHR_LOGFILE_PATH. In the end, we dump information about all the TIDs, UIDs and related GRs to NSHR_DUMPFILE_PATH. The small problem we had was that LP tool uses it's own buffering and DynamoRIO uses it's own. Thus, before asking LP tool to do the calculation we flush the DynamoRIO's buffers and at the end of LP tool's calculations, we flush LP tool's buffers.

3.3.6 Features

To help with debugging, we have implemented two custom DynamoRIO annotations that can be used directly from the client code. One of them, called *taint annotation*, can be used to manually taint some address in the application. We usually used this for tainting variables for debug purposes. Another

annotation, called *dump taint annotation* can be used to dump taint information (TID) of the variable. This, again, can be used to inspect variable's taint metadata at any point in the code, for debugging purposes.

Another interesting feature that we have implemented is functionality to ignore some functions. This can be useful when, for example, we know that some function is not interesting from the tainting perspective and we want to skip processing it for performance reasons. Another possible case when it may come in handy is when the function is too complicated (e.g. using a lot of bitwise operators or Intel MMX instructions) or leading to false positives and we decide to lose some precision by ignoring taint propagation t through this function. We have used this feature when analyzing OpenSSL library for famous Heartbleed vulnerability, which we will talk about in the *Results* section. We have ignored, for example, the encoding function which is written in assembly and is very complicated.

The *drsyms* DynamoRIO extension provides the functionality of reading symbol information. We have used this feature to dump the source code file and line number if the tool detects the vulnerability. Thus, it's a good idea to compile client with -g flag. We have also used this feature to provide more informative logs about every jump that happens in the application. However, this is a very expensive operation and is disabled by default. It can be enabled by defining DBG_PARSE_JUMPS. To improve the speed, we never decode this information about the same location twice. When we encounter a jump to some location for the first time, we decode and print the corresponding information. Plus, we store information that we have decoded this location in a special hashtable. Then, on every next jump, we first check if we have already decoded the target location. If we have, we just log the corresponding memory address. Then, the decoded information can easily be found in logs by tracing the first encounter of the target location there. This is a huge improvement because jumps to the same location are very common in code, mostly for frequently used functions and loops.

Apart from this define, we provide two additional defines to enable/disable debug features. One of them, called 'CHECKS', enables various assertions that we do in the application. Those assertions are used to make sure that everything is happening as we have planned and can be disabled in non-debug builds. Another define, 'DBG_PASS_INSTR', adds one more parameter to each clean call: object encoding the instruction. This is needed to log instruction location with each executed instruction or when the vulnerability is detected.

3.3.7 Limitations

In the first version we didn't have threading support. If the tool detects calls to any thread-related function (from *pthreads* library in our case), it immediately warns us about it and stops executing. We decided not to implement support for threading in the first version because most of the libraries that we were aiming to test our tool on either doesn't support threading or it can be turned off while compiling.

Another limitation of our tool is that we don't support instruction set extensions, like SSE.

3.3.8 Optimizations

One of the problems that we usually face is that sometimes GCC does optimizations that are very hard to analyze. For example, since integer division is a very slow operation for processors, compilers have found a way to implement division by a bunch of multiplication and shift operations if the divisor is known at the time of compilation. This can, in some cases, lead to wrong results by our tool. Other common optimizations that can be confusing for our tool are bit twiddling hacks. For example, when the user wants to compute string length with *strlen* function, the most straightforward and not so fast way to do so would be to iterate on each character and search for null character. However, there is a more efficient way to do it by jumping over four bytes at a time and using a special bit-twiddling hack to check if given word of four bytes has zero byte in it. The check looks like this:

```
#define haszero(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
```

For stated reasons, to get the best results we recommend to compile an application with -O0 before running with our tool.

Since our tool is quite heavy we have spent some time thinking about what are the possible optimizations we could do to improve the performance. One of the optimizations that we came up with is that the tool starts analyzing instructions in the client from the very beginning, that is, even before it jumps to main. Actual entry function of the executables on Linux is _start, and it does the whole bunch of initializations before it jumps to main. We can safely ignore this part, because, there is no taint in the client before jumping to main and thus, we cannot propagate anything. Thus, when the tool starts, it searches for the main function in the application and stores its address. Then, before we first jump to it, the tool executes in special initialization mode which basically does nothing except waiting for the instruction that jumps to main. When we jump to main we introduce our first taint by tainting command-line arguments (we should also taint environment variables but we don't have to, because we have added getenv function to *Skipper*'s list and we taint its return value) and we proceed with normal mode from then on.

3.3.9 Debugging

Debugging was the most time-consuming part of this project. Since the project is written in a C programming language, the tool was prone to many kinds of errors, like memory leaks and buffer overflows. Apart from that, we could not use any commonly used tools (like Valgrind or ASAN) to help with debugging memory corruptions because of the complex structure that DynamoRIO

has. Thus, our primary tool for debugging was logging. As you can see, if you turn on DUMP logs in all modules, log file easily becomes several gigabytes. We log everything in DUMP mode so that when an error happens, we have enough information about everything from logs that we can successfully track down and fix the bug that caused the error.

DynamoRIO, however, offers some help in DEBUG mode. First of all, it has an internal memory management system that warns if there was any memory leak after the tool has exited. It doesn't show any more information except the size of the leak but it was still very useful. Apart from that, if the tool crashes, DynamoRIO prints a backtrace of the tool, which is also very useful for debugging.

Chapter 4

Experimental results

In this section, we will talk about what kind of experimentation we did with our tool and what results we got. Our aim was at least to detect the Heartbleed vulnerability (Found under ID CVE-2014-0160 at National Vulnerability Database (NVD)), and possibly some other, new vulnerabilities in the existing libraries. Thus, we will also discuss Heartbleed vulnerability and show how our tool detects it.

4.1 Testing and Statistics

To test KATU for the correctness, we have developed a number of test applications and a script that runs the test applications with the tool and checks for both false positives and false negatives. With test applications we test various cases for all parts of taint analysis, including taint creation (tainting environment vars, command-line arguments, data read from files or sockets), taint propagation (via assembly instructions or libc functions) and taint checking (accessing by unbounded variable, complicated scenarios for LP). Apart from those tests, we ran regression tests of *libxml, binutils* and *libjpeg* with our tool. Tests on *libxml* were about parsing the XML files. *Binutils* consists of a bunch of binary tools, like *ls, as, objdump* and so on. The source code came with regression tests that only tested some of them. We took the biggest one, that was testing demangling functionality of *objdump*. Demangling means decoding the name of the symbol that got mangled by C++ compiler (like _ZNK6Object10metaObjectEv) back to human-readable form (Object::metaObject() const). The tests of *libjpeg* read and decode the jpeg images.

Those regression tests made it possible to detect and fix some of the very obscure bugs in KATU. Also, they served as good sources of real-life examples to get various statistical insights. For all three benchmarks corresponding execution times with and without the tool look like this:

	libjpeg	binutils	libxml
Native execution time	0.0062s	0.0048s	0.025s
Execution time with KATU	32.162s	20.767s	25.327
Approximate slowdown	5187	4326	1013

As you can see, the slowdown can be up to 5200 times. The major contribution to this slowdown are given factors:

1. Inserting clean calls for each instruction incurs huge slowdown because on each clean call DynamoRIO has to save application state information, set up call arguments and do the call, which means it has to do lots of instructions for each instruction.
2. We process taint on byte level. That means, doing one move from 64-bit register costs us 8 operations, each itself constituting of several instructions.
3. Some functions in libc do things like copying chunk of memory. Such functions are implemented in a highly efficient way in libc, but when we are processing such function we have to do a regular, full-sized loop to copy taint information of all the bytes participating in a copy.

Now let's take a look at statistics of instructions which can be found in Figure 4.1

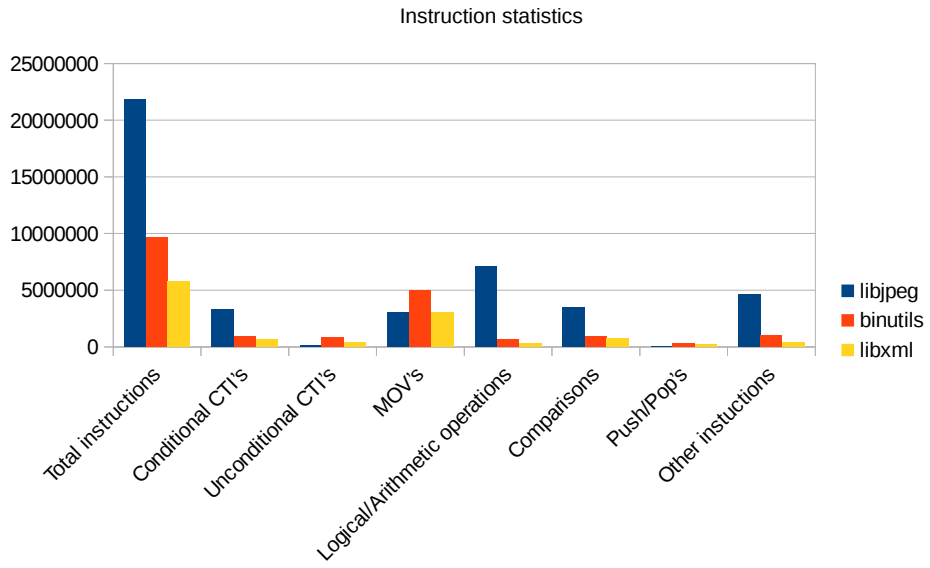


Figure 4.1: Instruction statistics, absolute numbers

To have a deeper insight, let's also take a look at the same statistics from a different perspective in Figure 4.2

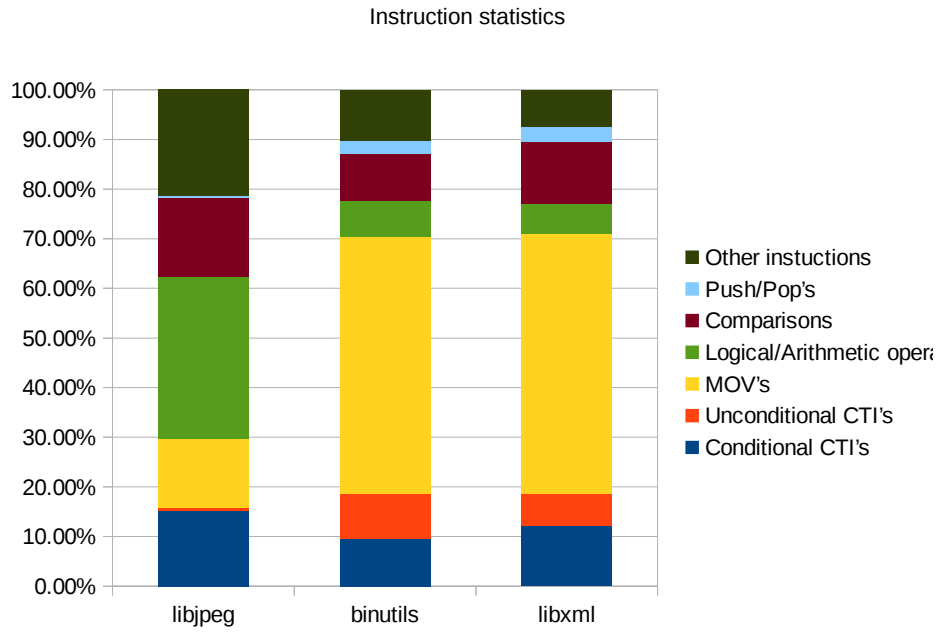


Figure 4.2: Instruction statistics, distribution

Libjpeg, that has the highest slowdown, has significantly more logical and arithmetic operations than the other two. This is because libjpeg has complicated compression algorithms built inside that do a lot of arithmetic and logic operations. Those operations can be costly to instrument, especially if the operands are tainted, because then we'll have to create and initialize new TID objects. Parsing XML and demangling, on the other hand, don't need to do many computations. They mostly consist of mov's, jumps and comparisons. *Libxml* has more comparisons and conditional jumps then *binutils*, which means that it either does more iterations in loops or has more if statements.

Since the distribution of instruction types look quite similar in *libxml* and *binutils*, and still, they have a large difference in slowdown, let's take a look at Figure 4.3 to see what functions from libc were they mostly using, according Skipper:

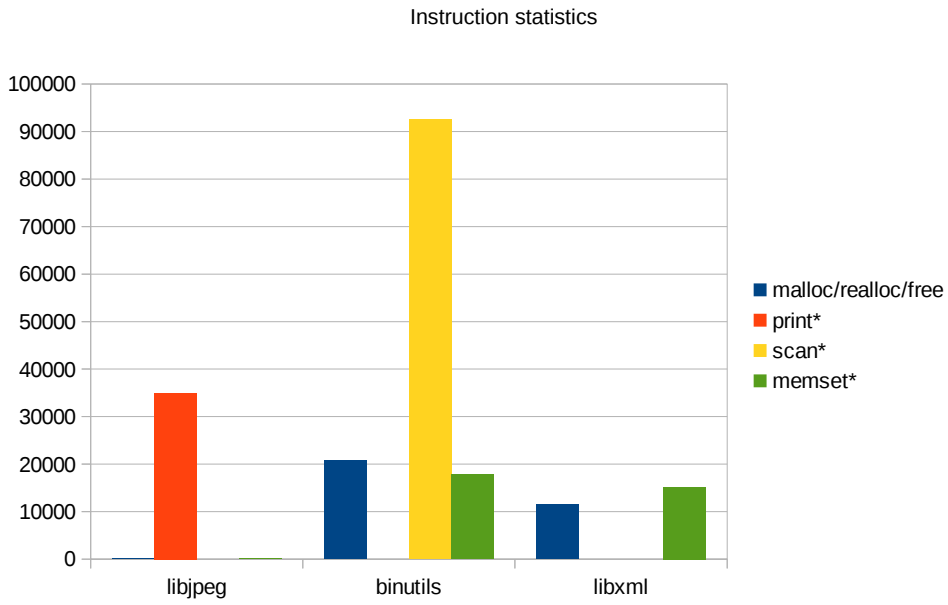


Figure 4.3: Usage of functions from libc.

Note that here, in *print** we have united the whole family of functions, like *printf, putchar, puts* and so on. Same goes for *scan** (*scanf, getchar* etc) and *memset* (*memcpy, strcpy, memove* etc). *Binutils* has used libc much more frequently than the rest two. This is because test application takes input from *stdin* via *getchar* and the file that contains the test data is 91KB. *Libjpeg* has used *printf** many times because it has to output decoded image in BMP format.

Apart from those statistics, we have also investigated KATU's resource usages with profiling tools. We have used OProfile's system-wide profiling functionality. It works by using underlying Linux Kernel Performance Events Subsystem. Reports can be seen in Figure 4.4.

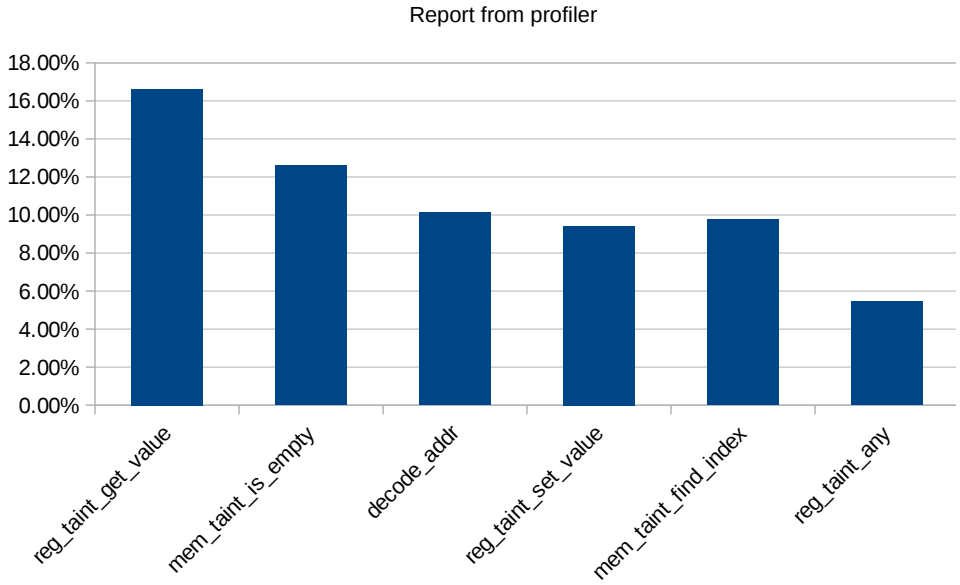


Figure 4.4: Reports from the profiler.

According to the picture, up to 17% of the time the tool is computing *reg_taint_get_value*, which returns TID for the given registry and index. A function is very straightforward:

```
#define REGINDEX(reg)                    (reg_mask_index[reg])
#define REGSTART(reg)                    (reg_mask_start[reg])

int64_t reg_taint_get_value(int reg, int offset)
{
  return taint_reg_.value[REGINDEX(reg)][REGSTART(reg) + offset];
}
```

*taint_reg_* here is a shadow object of the registry set. Thus, the functional complexity doesn't impose much slowdown, but the frequency that the function gets executed at does. OProfile showed us that 60% times it gets executed from *reg_taint_any*, function which checks if any byte in the registry is tainted. This function gets executed at the beginning of some taint propagation functions and serves as a quick exit if the operands and destination are not tainted. In *Future work* section we will talk about possible improvements of this function.

Next comes *mem_taint_is_empty*, a function that is used internally when searching in shadow memory hashmap. Thus, up to 13% of the whole execution time is spent in searching for taint information of some byte in memory.

Next one is *decode_addr*, a function that is used to calculate absolute memory address if scaled indexed addressing mode is used. This function basically gets registry values from saved context and calculates *base_reg + scale*index_reg + disp*. This amounts to 10% of the whole execution time.

Next one is *reg_taint_set_value*, which sets TID to given registry and index. Implementation is again, very straightforward:

```
void    reg_taint_set_value(int reg, int offset, uint64_t value)
{
  taint_reg_.value[REGINDEX(reg)][REGSTART(reg) + offset] = value;
}
```

Functions that call *reg_taint_set_value* are mostly taint propagation functions.

Next one is *mem_taint_find_index*. This is the main function that searches in the shadow memory and calls *mem_taint_is_empty*. Since this function attributes to 10% of the execution time, we have already spent up to 23% of whole execution time in searching for TID of memory locations.

Next one is *reg_taint_any*, which we have already mentioned. This function can be improved and added to all taint propagation functions to achieve faster execution times. Apart from that, as we have seen, currently it works by manually checking all bytes of the register. All the remaining functions amount to less than 4% for whole execution time, each.

These statistics have shown us what are the most frequently used instructions, most frequent calls to libc and what are the major causes of having such a huge slowdown. Unfortunately, we could not detect any new bug in those libraries while running regression tests with KATU. Also, since we are not aware of any tool that does the similar thing as we do, we could not do any comparison.

4.2 HeartBleed

The Heartbleed Bug is a serious vulnerability in the OpenSSL library. OpenSSL is an open-source library, written in C programming language, that provides an implementation of cryptographic protocols like SSL and TLS. The vulnerability allows stealing the protected information from the systems having a vulnerable version of OpenSSL. At the time of disclosure of the bug (April 7, 2014), around 17% (approx. half million) of secure web servers certified by trusted authorities were vulnerable to the attack. Thus, this vulnerability had a huge impact on the whole Internet. That's why we decided to investigate it deeper and check if our tool would have detected it.

The vulnerability is the result of the improper bounds checking on the input, which happens in TLS heartbeat extension's implementation. The extension is used to check if the communication between two clients is still 'alive'. It consists of two messages: *HeartbeatRequest* and *HeartbeatResponse*. The protocol itself

is simple: One side sends a *HeartbeatRequest* message to the other side. The other side should reply with *HeartbeatResponse* message to make a successful Heartbeat and keep the connection alive. If there is no response within some time, the connection is terminated. The following code is from OpenSSL's implementation of sending *HeartbeatRequest*:

```
unsigned int payload = 18; /* Sequence number + random bytes */
unsigned int padding = 16; /* Use minimum padding */

.......

/* Check if padding is too long, payload and padding
* must not exceed 2^14 - 3 = 16381 bytes in total.
*/
OPENSSL_assert(payload + padding <= 16381);
/* Create HeartBeat message, we just use a sequence number
* as payload to distuingish different messages and add
* some random stuff.
* - Message Type, 1 byte
* - Payload Length, 2 bytes (unsigned int)
* - Payload, the sequence number (2 bytes uint)
* - Payload, random bytes (16 bytes uint)
* - Padding
*/

buf = OPENSSL_malloc(1 + 2 + payload + padding);
p = buf;
/* Message Type */
*p++ = TLS1_HB_REQUEST;
/* Payload length (18 bytes here) */
s2n(payload, p);
/* Sequence number */
s2n(s->tlsext_hb_seq, p);
/* 16 random bytes */
RAND_pseudo_bytes(p, 16);
p += 16;
/* Random padding */
RAND_pseudo_bytes(p, padding);
```

The first byte in the message corresponds to packet identified (*TLS1_HB_REQUEST*). Next 2 bytes encode the size of the whole packet. Then comes the actual payload (2 bytes of sequence number plus 16 random bytes) and the padding at the end. As you can see, "payload and padding must not exceed 16381 bytes in total". The whole packet size is maximum 16KB, including message type and length bytes. However, this one has only 18 bytes of payload.

Next, let's take a look at the code that processes the received *HeartbeatRequest* and sends the *HeartbeatResponse* back:

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;

.....

if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;
    /* Allocate memory for the response, size is 1 bytes
    * message type, plus 2 bytes payload length, plus
    * payload, plus padding
    */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    bp += payload;
    /* Random padding */
    RAND_pseudo_bytes(bp, padding);
    r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
    .....
```

Here, the first thing we do is reading *hbtype* and *payload*. Then, if *hbtype* corresponds to *HeartbeatRequest*, we allocate memory for the *HeartbeatResponse*, set the identifier of the packet (*TLS1_HB_RESPONSE*) and copy the payload from *HeartbeatRequest*. Here is the problem. We copy from *HeartbeatRequest*'s payload to *HeartbeatResponse*'s payload the whole payload, which we assume has the size that was specified in *HeartbeatRequest*'s packet.

If *HeartbeatRequest*'s sender specifies the size that is larger than actual payload, an interesting thing happens: *memcpy* command on the receiving side, that was supposed to copy sender's payload to the response packet, copies extra bytes from the memory to the responded packet! Thus, by wrongly specifying the larger size of the packet, the sender has the ability to read the receiver's memory past the sent packet. Since this size parameter is 2-bytes, the sender can read up to 65KB of the memory! This is called buffer over-read vulnerability.

For our tool, this corresponded to having a 2-byte unbounded argument in *memcpy* and our tool was able to detect the vulnerability. But, we had to compile OpenSSL with O0, because, as we have already discussed, it can lead to false results sometimes. Apart from this, we have used KATU's functionality of ignoring selected functions and we have ignored analyzing execution of three following functions: *SSL_connect*, *SSL_accept* and *tls1_enc*. We did so because internally OpenSSL uses very complicated mechanisms to do the encoding and decoding of the messages. When doing *connect* or *accept*, it does manipulations with certificates, which are tainted and those manipulations lead to wrong results.

We have made those changes and wrote a small client/server applications that connect and exchange heartbeat requests. Our tool was able to detect the vulnerability when we ran the client with the tool.

Chapter 5

Related Work

What we have implemented is a dynamic vulnerability discovery tool. Generally, such tools are split into two different categories: ones with concrete execution and ones with symbolic execution. Dynamic concrete execution works by executing a program for given inputs, following a single path in CFG. Thus, those tools require test cases. Most common tools in this category are Fuzzers. They work by taking test cases and doing random mutations, trying to trigger a crash. Dynamic symbolic execution, on the other hand, works by executing the application on *symbolic variables* in an emulated environment. As the execution proceeds, they track all the constraints applied to the variables. When the conditional branch is encountered, execution follows both paths by forking the saved constraints and adding branch-related constraints to both cases: constraint satisfying the conditional jump to execution that follows the branch-taken path and constraint not satisfying the conditional jump to execution that follows the branch-not-taken path. Those tools have much higher semantic insight than concrete execution ones, but they are also much slower and suffer from path explosion problem: the number of paths increases exponentially for each conditional branch instruction encountered in any path.

Our tool lies somewhere in the middle. Just like dynamic concrete execution tools, we also follow a single path in the CFG, one that application runs through for given inputs. Thus, as we have just mentioned, we need test cases. On the other hand, just like dynamic symbolic execution tools, we remember the constraints applied to our variables. However, we only remember the simplified version of constraints. Thus, we have taken ideas from both categories of tools.

Clause et al [5] have proposed a technique that has inspired us. They have developed a DTA tool that tries to detect memory errors. They mark two kinds of data: memory in the data space and pointers. When a new memory is allocated, the allocated memory is tainted with some mark t. The pointer that points to this memory is also marked with t. Thus, they keep track of pointer - block of memory correspondence. Then they propagate the taint marks and when the memory is accessed they check if memory area and pointer have the same mark. If they don't, it means that pointer is accessing the out-of-bounds

memory and they have found a vulnerability. For statically allocated variables, they intercept function entry(for local variables) and program entry (for global variables) events and do the taint markings.

Described tool, unlike ours, can detect off-by-one errors because it knows the exact bounds of the allocation that the pointer is pointing to. In the beginning, we wanted to take the same idea and just extend it to detecting possible out of bound memory accesses by remembering the exact constraints applied to the given path. This tool then would have been able to detect not only unbounded accesses but incorrectly bounded accesses as well. However, things got very complicated because then on every memory access we would have to do value-set analysis instead of simple bounds checking and value-set analysis is hard.

Just like our tool, this tool also ignores cases like XOR'ing two tainted variables.

Chapter 6

Future Work

The biggest amount of work that we are planning to do in the next versions is trying to improve performance, since as we have already stated we have up to 5200 times slowdown in some cases.

Big improvements can be done on taint propagation side. For example, in the current version, for each *mov* instruction of 64-bit register to register, we iterate over all bytes of the source register and copy the taint information. This, however, can be skipped if we knew in advance that none of the registers are tainted. We have a function, (*reg_taint_any*) that serves a similar purpose and gets called from some taint propagation functions. In the current implementation, it's slow. As we have stated, currently it just iterates over all bytes and checks their taints. Instead of iterating over all bytes we can have a separate variable, a bitmap for each 64-bit registers, each bit indicating whether or not the corresponding byte is tainted. When we will taint, for example, EAX, we will just *OR* the corresponding bitmap (RAX) with 0xF. When we will need to see if the register is tainted we will clear non-necessary bits with an *AND* operation and check if the result is more than zero. For memory locations, we can have a separate shadow memory object, where each byte will be mapped to a single bit. Updating and checking taint information will work in a similar way as with registers. This is one example from a lot of other possible improvements, which will greatly increase the speed of the tool.

In the current version, when the vulnerability is detected we only print the address of the last instruction. However, if it's in one of the commonly used functions it's not very helpful for debugging the application. Instead, it would be much more beneficial if we could print the whole stack trace. DynamoRIO doesn't have a functionality of decoding the stack trace and we didn't implement it in KATU in the first version.

Another improvement that is planned for the next versions is related to LP solver. In the current version, we have used LP solver to solve the boundedness problem, which was not absolutely necessary. We have decided to use an LP solver because it was an easy solution that would do the job. Otherwise, we would have had to develop our own algorithm to solve the boundedness and this

algorithm could have had some complications. Solving boundedness is a very narrow, less complicated subset of general problems LP solver deals with. For the next version, we have planned to get rid of LP solver and implement our own algorithm that will solve the boundedness problem under given constraints.

Another big improvement can be achieved by detecting *Terminal functions*. We define *Terminal function* as a function that doesn't access any global variables and calls only other terminal functions. If we could identify the terminal functions, we can directly ignore execution of the whole function when the arguments are untainted because it won't modify any global data and the returned value will be untainted. Thus, that function's execution won't change anything in our shadow memory. This is a very promising idea because a lot of functions in real-world applications would fit into this category and it could improve the performance of the tool a lot.

Since KATU reasons only about single CFG path, it won't be able to recognize terminal function before we cover all the possible code inside. Thus, in order to recognize terminal functions, we would need some additional tool that would do the static analysis. Also, we can ask developers to identify terminal functions with special pragma directives. This is the subject of our future research.

Bibliography

[1] Creating guard pages. https://docs.microsoft.com/en-us/windows/desktop/memory/creating-guard-pages (May. 2018).

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[4] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[5] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 284–292, New York, NY, USA, 2007. ACM.

[6] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[7] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreÉ analyzer. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, pages 21–30, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[9] Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. The meaning of memory safety. *CoRR*, abs/1705.07354, 2017.

[10] Ulrich Drepper. Elf handling for thread-local storage. 2013.

[11] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[12] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[13] Donny Hubener. Understanding seh (structured exception handler) exploitation. 2009.

[14] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[15] Richard W M Jones, Paul H J Kelly, Most C, and Uncaught Errors. Backwards-compatible bounds checking for arrays and pointers in c programs. In *in Distributed Enterprise Applications. HP Labs Tech Report*, pages 255–283, 1997.

[16] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[17] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[18] Dr. Hector Marco-Gisbert and Dr. Ismael Ripoll-Ripoll. Exploiting linux and pax aslr's weaknesses on 32- and 64-bit systems. 2016.

[19] Paul Mutton. Half a million widely trusted websites vulnerable to heartbleed bug. `https://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html`, April 2014.

[20] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[21] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[22] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[23] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 73–85, New York, NY, USA, 2009. ACM.

[24] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *IN PROCEEDINGS OF THE 11TH ANNUAL NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*, pages 159–169, 2004.

[25] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[26] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[28] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[29] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on*

Information Systems Security, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. 2018.

[31] Eugene H. Spafford. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, January 1989.

[32] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[33] Ondrej Vasık and Kamil Dudka. Common errors in c/c++ code and static analysis.